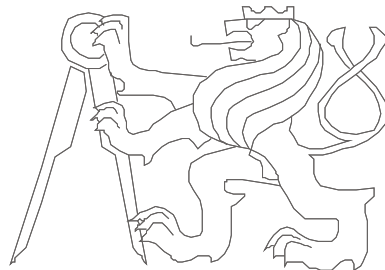


Computer Architectures

Real Numbers and Computer Memory

Pavel Píša, Richard Šusta, Petr Štěpán

Michal Štepanovský, Miroslav Šnorek



Czech Technical University in Prague, Faculty of Electrical Engineering

English version partially supported by:

European Social Fund Prague & EU: We invests in your future.





APO at Dona Lake
84° 28' 45" E, 28° 29' 52" N, 4038m, 2019-11-28
APO at InstallFest (<https://installfest.cz/>)
2021-03-06 via BigBlueButton running at
50°4'36.682"N, 14°25'4.116"E
QtMIPS Hands on Session to Understand
Computer Architectures and Discuss Its
Teaching
Embedded Linux, FPGA and Motion Control
Hands-On

Classification and Conditions to Pass the Subject

Conditions for assessment:

Category	Points	Required minimum	Remark
4 homeworks	36	12	3 of 4
Activity	8	0	
Team project	24	5	
Sum	60 (68)	30	

Grade	Points range
A	90 and more
B	80 - 89
C	70 - 79
D	60 - 69
E	50 - 59
F	less than 50

Exam:

Category	Points	Required minimum
Written exam part	30	15
Oral exam part	+/- 10	0

Lecture Contents

- Number representation in computers
 - numeral systems
 - integer numbers, unsigned and signed
 - boolean values
- Basic arithmetic operations and their implementation
 - addition, subtraction
 - shift right/left
 - multiplication and division
- Real numbers
 - representation
 - computation

Non-positional numbers 😊



<http://diameter.si/sciquest/E1.htm>

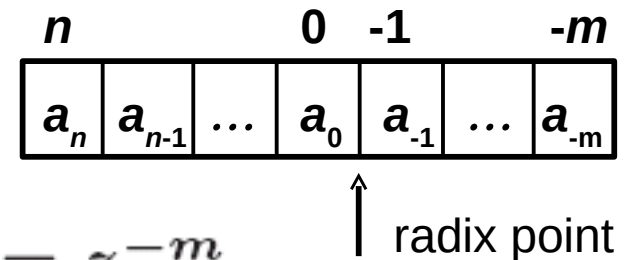


10, 100, 1000, 10000, 100000, 1 million

The value is the sum: 1 333 331

Terminology Basics

- Positional (place-value) notation
- Decimal/radix point
- z ... base of numeral system
- smallest representable number $\epsilon = z^{-m}$
- **Module** = Z , one increment/unit higher than biggest representable number for given encoding/notation
- **A**, the representable number for given n and m selection, where k is natural number in range $\langle 0, z^{n+m+1} - 1 \rangle$
- The representation and value



$$0 \leq A = k \cdot \epsilon < Z$$

$$A \sim a_n a_{n-1} \dots a_0, a_1 \dots a_{-m}$$

$$A = a_n z^n + a_{n-1} z^{n-1} + \dots + a_0 + a_1 z^{-1} \dots a_{-m} z^{-m}$$



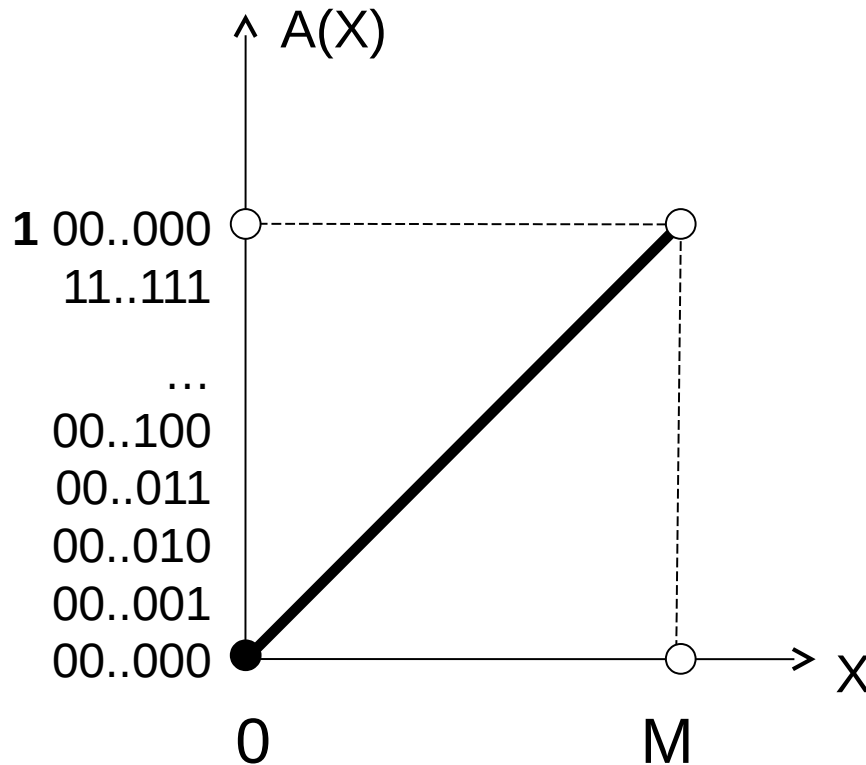
Unsigned Integers

Language C:
`unsigned int`

Integer Number Representation (unsigned, non-negative)

- The most common numeral system base in computers is $z=2$
- The value of a_i is in range $\{0,1,\dots,z-1\}$, i.e. $\{0,1\}$ for base 2
- This maps to true/false and unit of information (bit)
- We can represent number $0 \dots 2^n-1$ when n bits are used
- Which range can be represented by one byte?
1B (byte) ... 8 bits, $2^8 = 256_d$ combinations, values $0 \dots 255_d = 0b11111111_b$
- Use of multiple consecutive bytes
 - 2B ... $2^{16} = 65536_d$, $0 \dots 65535_d = 0xFFFF_h$, (h ... hexadecimal, base 16, a in range 0, ... 9, A, B, C, D, E, F)
 - 4B ... $2^{32} = 4294967296_d$, $0 \dots 4294967295_d = 0xFFFFFFFF_h$

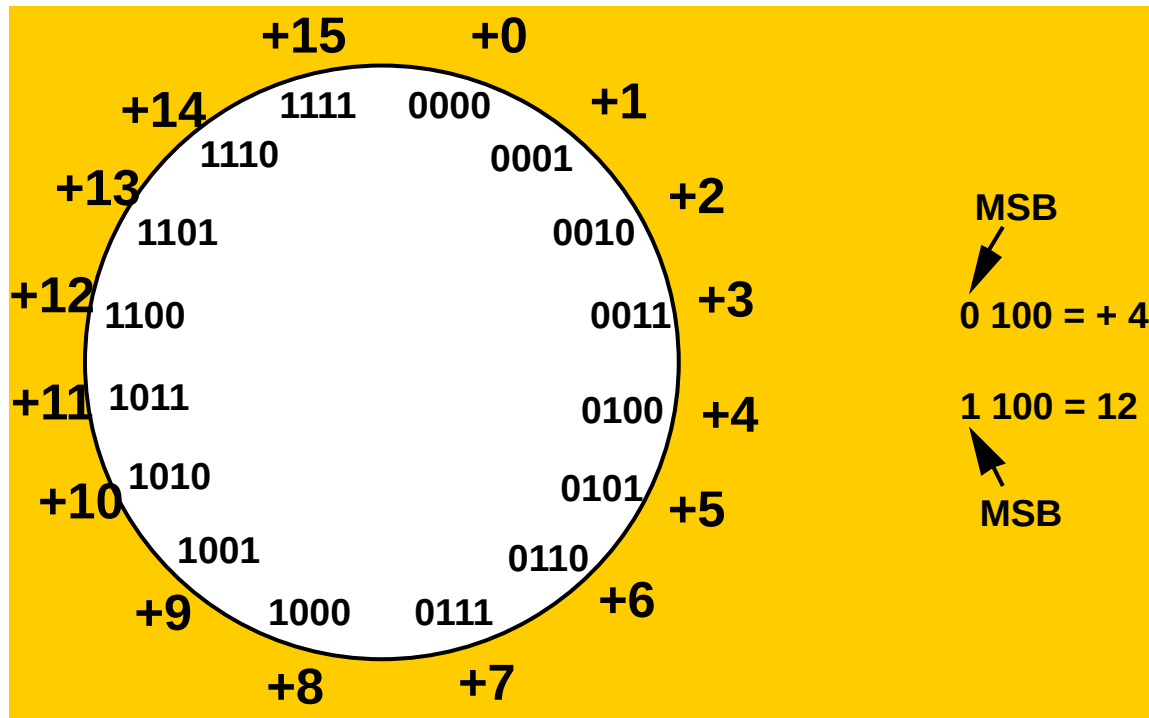
Unsigned Integer



binary value	hex	unsigned int
00000000	00	$0_{(10)}$
00000001	01	$1_{(10)}$
⋮		⋮
01111101	7D	$125_{(10)}$
01111110	7E	$126_{(10)}$
01111111	7F	$127_{(10)}$
10000000	80	$128_{(10)}$
10000001	81	$129_{(10)}$
10000010	82	$130_{(10)}$
⋮		⋮
11111101	FD	$253_{(10)}$
11111110	FE	$254_{(10)}$
11111111	FF	$255_{(10)}$

Unsigned 4-bit numbers

Assumptions: we'll assume a 4 bit machine word



■ Cumbersome subtraction

[Seungryoul Maeng: Digital Systems]



Signed Numbers

Language C:

`int`

`signed int`

Signed Integer Numbers

- Work with negative numbers is required for many applications
- When appropriate representation is used then same hardware (with minor extension) can be used for many operations with signed and unsigned numbers
- Possible representations
 - sign-magnitude code, direct representation, sign bit
 - two's complement
 - ones' complement
 - excess-K, offset binary or biased representation

Two's Complement (Complement to Module)

- The most frequent code
- **The unsigned sum of two opposite numbers representations with the same absolute value is 00000000H!**
- Transform to the representation of N digits:

$$A(X) = X \quad \text{iff } X \geq 0$$

$$A(X) = z^N - |X| \quad \text{iff } X < 0$$

$$A(X) = X \bmod z^N$$

- The name is misleading, it is module (z^N) complement actually
- Negative value can be encoded as digit by digit $(z-1)-a_i$ and then add 1
- For $z=2$, bit by bit complement ($0 \rightarrow 1, 1 \rightarrow 0$) and addition of 1

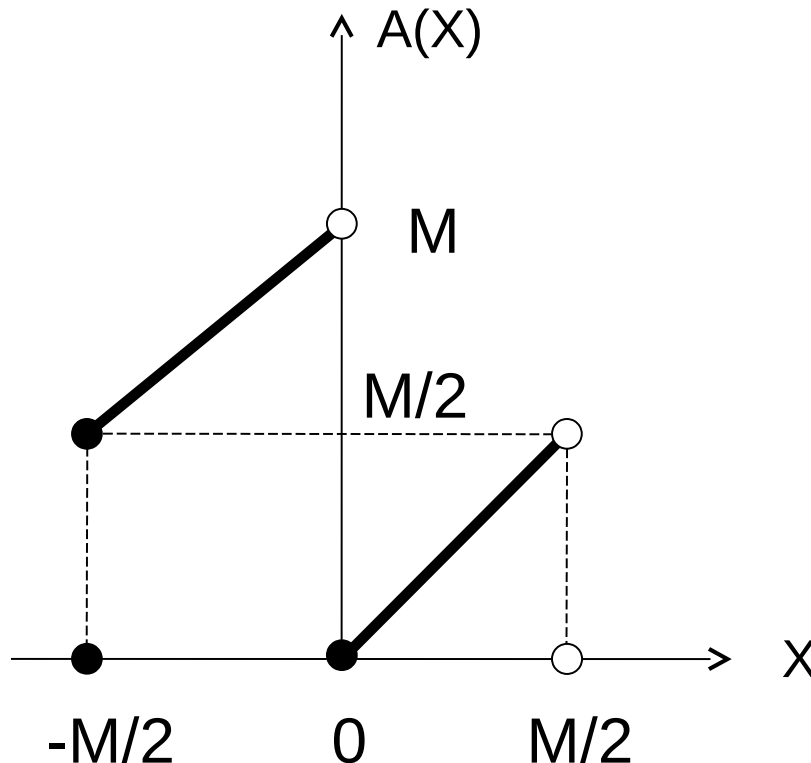
Decimal value	2^4	D(A)	4 bit two's complement
6	16	6	0110
-6	16	10	1010

Two's Complement

For N bits used for representation

$M=2^N$, represented value range:

$\langle -2^{N-1}, 2^{N-1} - 1 \rangle$



Binární hodnota	Dvojkový doplněk
00000000	$0_{(10)}$
00000001	$1_{(10)}$
⋮	⋮
01111101	$125_{(10)}$
01111110	$126_{(10)}$
01111111	$127_{(10)}$
10000000	$-128_{(10)}$
10000001	$-127_{(10)}$
10000010	$-126_{(10)}$
⋮	⋮
11111101	$-3_{(10)}$
11111110	$-2_{(10)}$
11111111	$-1_{(10)}$

Two's Complement - Examples

- **Advantages**

- Continuous range when cyclic arithmetic is considered

- Single and one to one mapping of value 0

- Same HW for signed and unsigned adder

- **Disadvantages**

- Asymmetric range, there is more negative numbers than positive

- **Examples:**

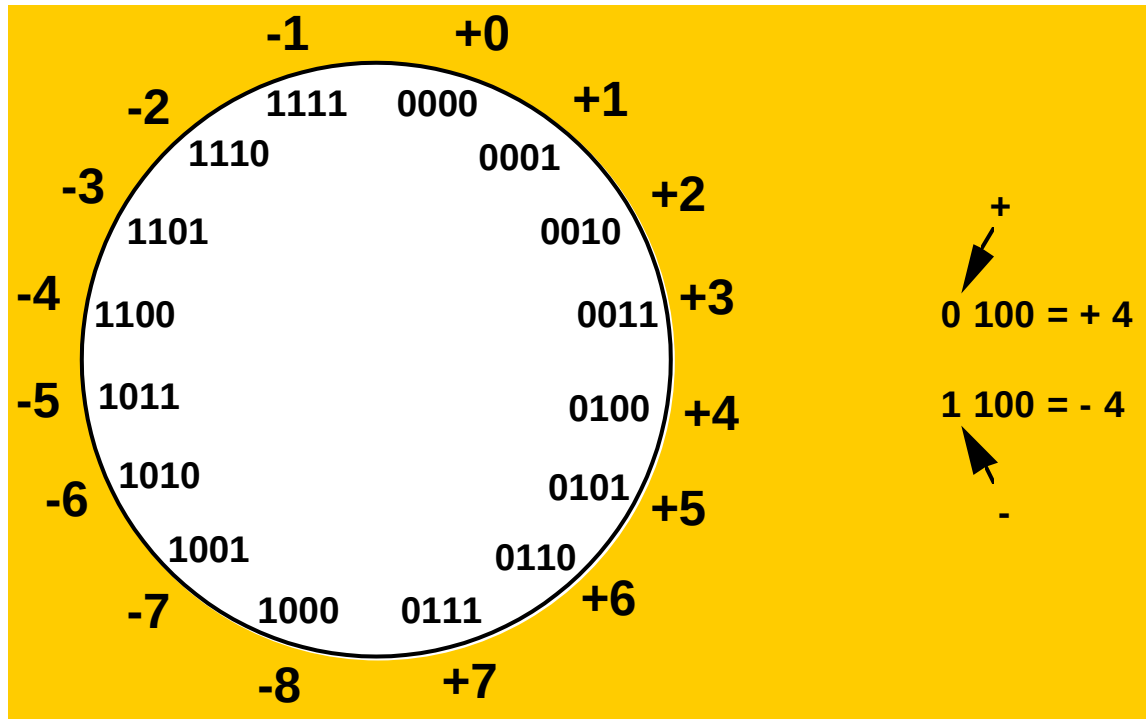
- $0_D = 00000000_H,$

- $1_D = 00000001_H, \quad -1_D = FFFFFFFF_H,$

- $2_D = 00000002_H, \quad -2_D = FFFFFFFE_H,$

- $3_D = 00000003_H, \quad -3_D = FFFFFFFD_H,$

Twos Complement (In Czech: Druhý doplněk)



Only one representation for 0

One more negative number than positive number

[Seungryoul Maeng:Digital Systems]

Two's Complement and the C Language

- Two's complement is most used signed integer numbers representation in computers
- Complement arithmetic is often used as its synonym
- “C” programming language speaks about integer numeric type without sign as *unsigned integers* and they are declared in source code as:
 - `unsigned int.`
- The numeric type with sign is simply called *integers* and is declared as:
 - `signed int`
 - `int`
- Considerations about overflow and underflow are described later.

Two's Complement and the C Language

- The size of integers in C:

Počet bajtů	Deklarace v C	Rozsah	
		Min	Max
1	unsigned char	0	255
1	char	-128	127
2	unsigned short int	0	65,535
2	short int	-32,768	32,767
4	unsigned (long) int	0	4,294,967,295
4	(long) int	-2,147,483,648	2,147,483,647
8	unsigned long (long) int	0	18,446,744,073,709,551,615
8	long (long) int	-9,223,372,036,854,775,808	+9,223,372,036,854,775,807

Two's Complement – Addition and Subtraction

- **Addition**

- $00000000\ 0000\ 0111_B \approx 7_D$ Symbols use: $0=0_H$, $0=0_B$

- $+ \underline{00000000\ 0000\ 0110_B} \approx 6_D$

- $00000000\ 0000\ 1101_B \approx 13_D$

- **Subtraction** can be realized as addition of negated number

- $00000000\ 0000\ 0111_B \approx 7_D$

- $+ \underline{FFFFFFF\ 1111\ 1010_B} \approx -6_D$

- $00000000\ 0000\ 0001_B \approx 1_D$

- Question for revision: how to obtain negated number in two's complement binary arithmetics?



Other Possibilities

Integer – Biased Representation

- Known as excess-K or offset binary as well
- Transform to the representation
$$A(X) = X+K \quad X \in \langle -K, 2^n - 1 - K \rangle \quad D(A) = A - K$$
- Usually $K=Z/2$
- Advantages
 - Preserves order of original set in mapped set/representation
- Disadvantages
 - Needs adjustment by -K after addition and +K after subtraction processed by unsigned arithmetic unit
 - Requires full transformation before and after multiplication

Addition and Subtraction for the Biased Representation

- Addition and Subtraction for Biased Representation:

$$A(X+Y) = (X+Y) + K = (X+K) + (Y+K) - K = A(X) + A(Y) - K$$

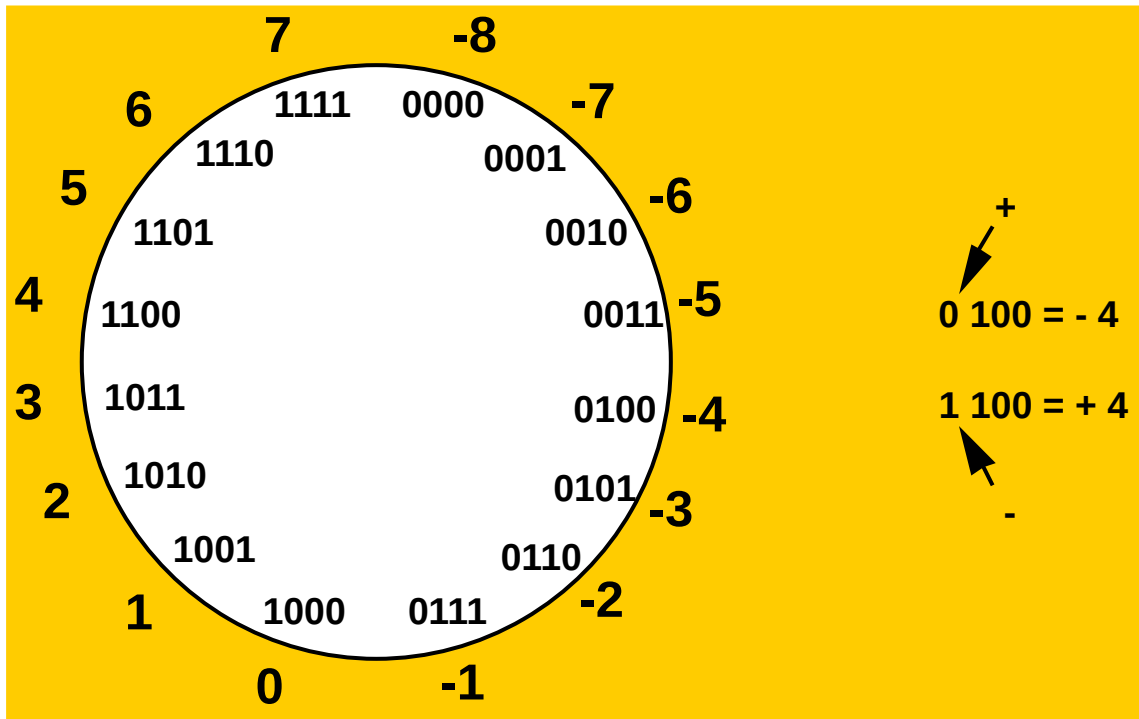
$$A(X-Y) = (X-Y) + K = (X+K) - (Y+K) + K = A(X) - A(Y) + K$$

- Multiplication is even worse:

$$\begin{aligned} A(X \cdot Y) &= (X \cdot Y) + K = (X+K) \cdot (Y+K) - (X+K+Y+K) \cdot K + K^2 + K \\ &= A(X) \cdot A(Y) - (A(x) + A(y)) \cdot K + K^2 + K \end{aligned}$$

- Overflow detection
 - for addition:
 - same sign of addends and different result sign
 - for subtraction:
 - signs of minuend and subtrahend are opposite and sign of the result is opposite to the sign of minuend

Excess-K, Offset Binary or Biased Representation



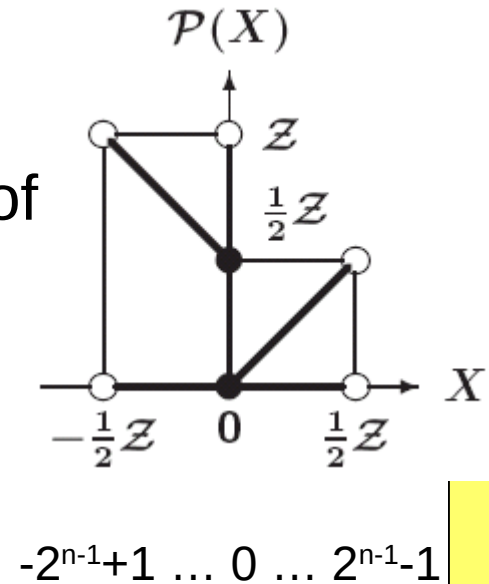
One 0 representation, we can select count of negative numbers -
used e.g. for exponents of real numbers..

Integer arithmetic unit are not designed to calculate with Excess-K numbers

[Seungryoul Maeng:Digital Systems]

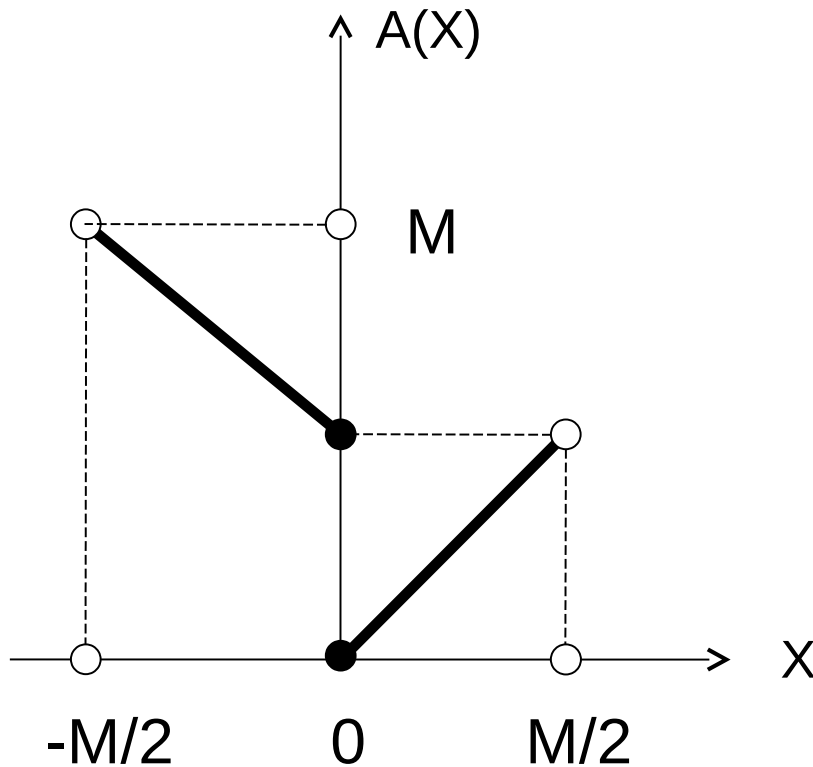
Integer – Sign-Magnitude Code

- Sign and magnitude of the value (absolute value)
- Natural to humans -1234, 1234
- One (usually most significant – MSB) bit of the memory location is used to represent the sign
- Bit has to be mapped to meaning
- Common use $0 \approx “+”$, $1 \approx “-”$
- Disadvantages:
 - When location is **k** bits long then only **k-1** bits hold magnitude and each operation has to separate sign and magnitude
 - Two representations of the value 0



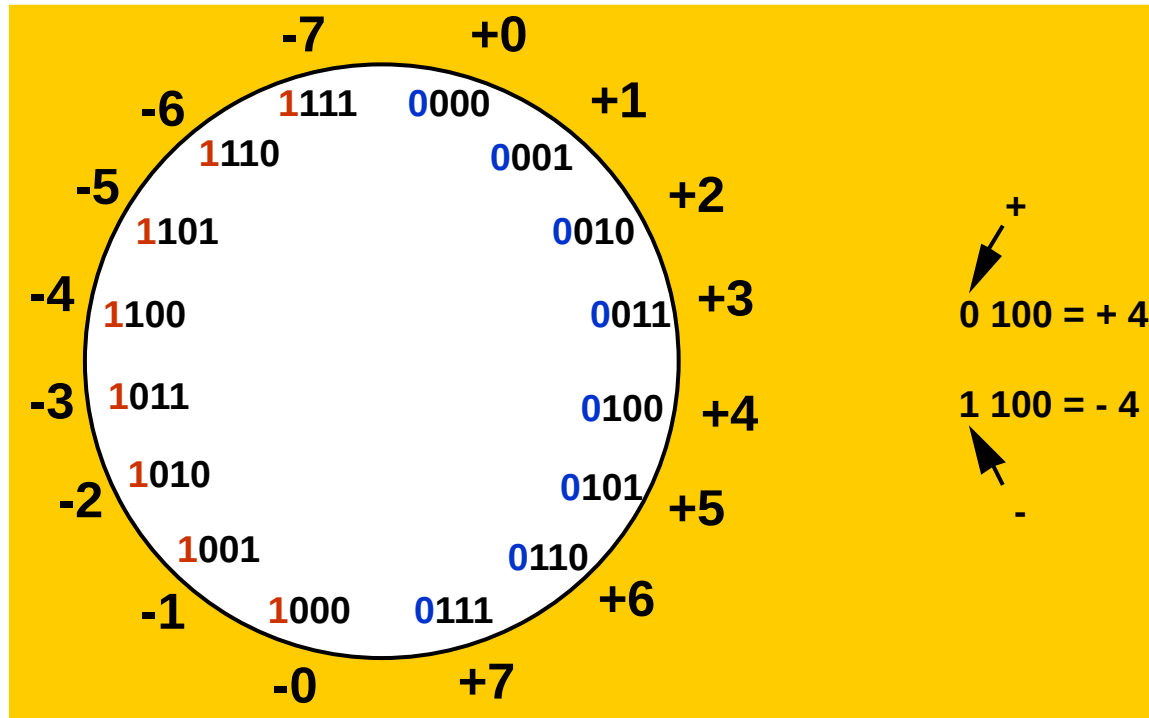
Sign and Magnitude Representation

$\langle -2^{N-1} - 1, 2^{N-1} - 1 \rangle$



Binary value	Code
00000000	$+0_{(10)}$
00000001	$1_{(10)}$
⋮	⋮
01111101	$125_{(10)}$
01111110	$126_{(10)}$
01111111	$127_{(10)}$
10000000	$-0_{(10)}$
10000001	$-1_{(10)}$
10000010	$-2_{(10)}$
⋮	⋮
11111101	$-125_{(10)}$
11111110	$-126_{(10)}$
11111111	$-127_{(10)}$

Sign and Magnitude Representation



- Cumbersome addition/subtraction
- *Sign+Magnitude usually used only for float point numbers*

[Seungryoul Maeng:Digital Systems]

Integers – Ones' Complement

- Transform to the representation with N digits

$$D(A) = A \quad \text{iff } A \geq 0 \quad -2^{n-1}+1 \dots 0 \dots 2^{n-1}-1$$

$$D(A) = z^N-1-|A| \quad \text{iff } A < 0 \quad (\text{i.e. subtract from all ones})$$

- Advantages

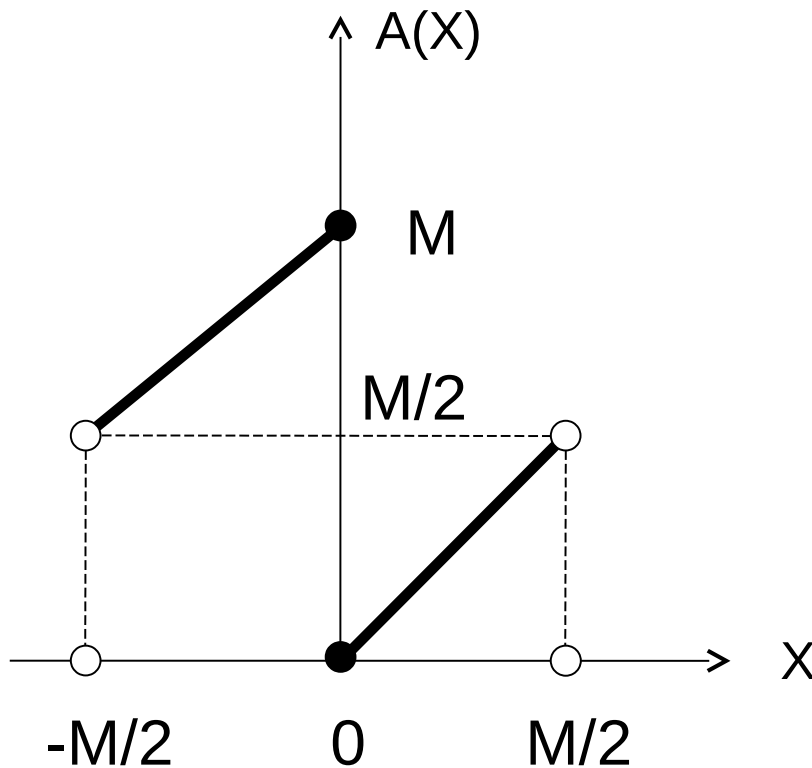
- Symmetric range
- Almost continuous, requires hot one addition when sign changes

- Disadvantage

- Two representations of value 0
- More complex hardware
- Negate (-A) value can be computed by bitwise complement (flipping) of each bit in representation

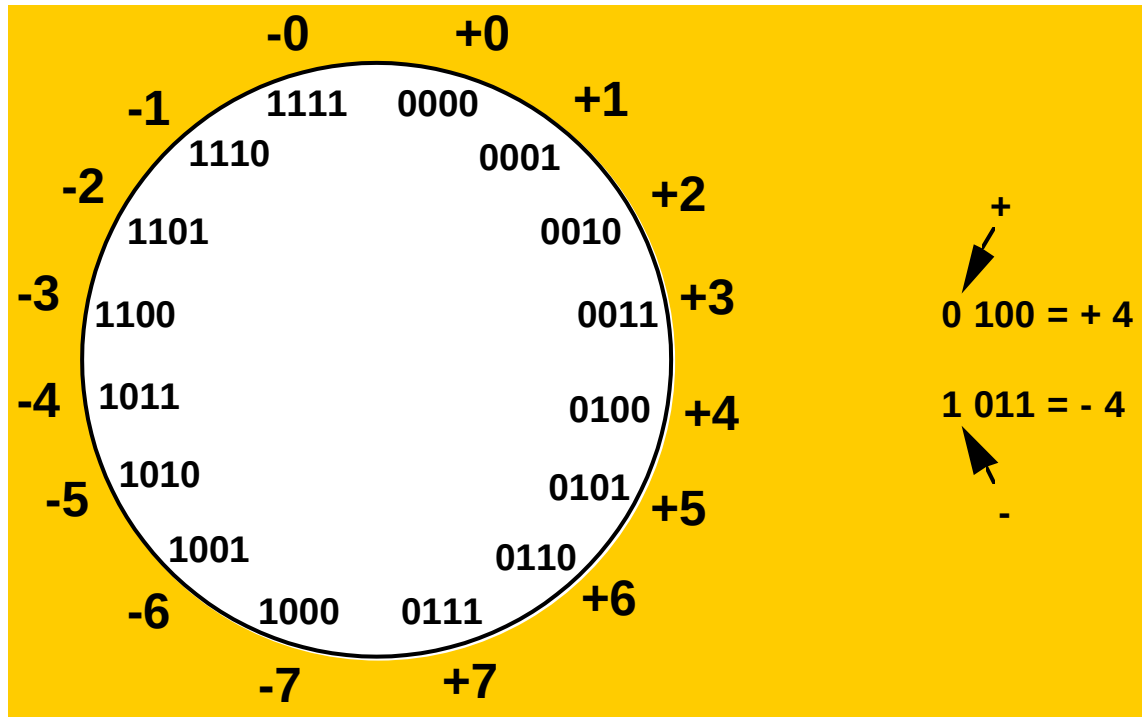
Ones Complement

$\langle -2^{N-1} - 1, 2^{N-1} - 1 \rangle$



Binary value	Code
00000000	$0_{(10)}$
00000001	$1_{(10)}$
⋮	⋮
01111101	$125_{(10)}$
01111110	$126_{(10)}$
01111111	$127_{(10)}$
10000000	$-127_{(10)}$
10000001	$-126_{(10)}$
10000010	$-125_{(10)}$
⋮	⋮
11111101	$-2_{(10)}$
11111110	$-1_{(10)}$
11111111	$-0_{(10)}$

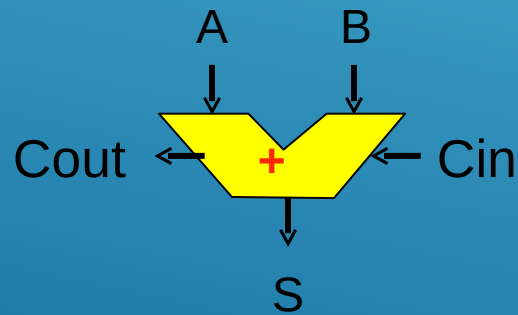
Ones Complement (In Czech: První doplněk)



Still two representations of 0! This causes some problems
Some complexities in addition, nowadays nearly not used

[Seungryoul Maeng:Digital Systems]

A	0	0	1	1	0	0	1	1
+B	0	1	0	1	0	1	0	1
Sum	00	01	01	10	00	01	01	10
+ Carry-In	0	0	0	0	1	1	1	1
CarryOut Sum	00	01	01	10	01	10	10	11



OPERATION WITH INTEGERS

Direct Realization of Adder as Logical Function

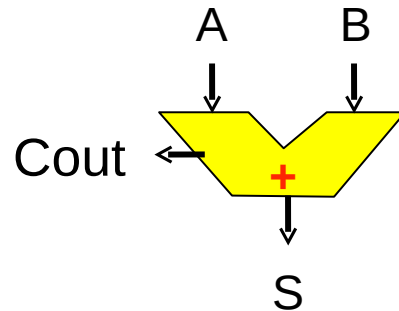
	Number of logic operations for calculating sum
bit width	
1	3
2	22
3	89
4	272
5	727
6	1567
7	3287
8	7127
9	17623
10	53465
11	115933

Complexity is higher than $O(2^n)$

*The calculation was performed by BOOM logic minimizer
created at the Department of Computer Science CTU-FEE*

1bit Half Adder

A	0	0	1	1
+B	0	1	0	1
Sum	00	01	01	10
Cout S	00	01	01	10

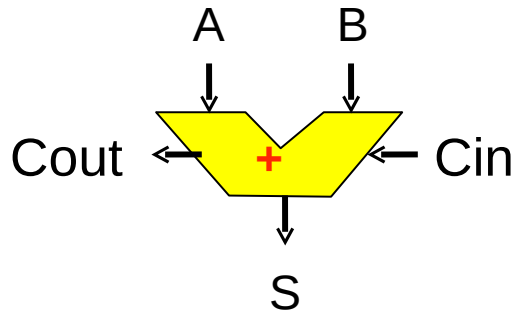


$$S = A \text{ xor } B$$
$$\text{Cout} = A \text{ and } B$$

1bit Full Adder

A	0	0	1	1	0	0	1	1
+B	0	1	0	1	0	1	0	1
Sum	00	01	01	10	00	01	01	10
+ Carry-In	0	0	0	0	1	1	1	1
Carry-Out S	00	01	01	10	01	10	10	11

2x Half Adder + 1x or:



$$S_1 = A \text{ xor } B$$

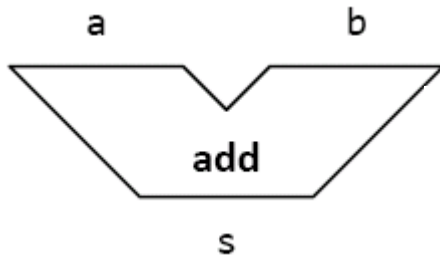
$$Cout_1 = A \text{ and } B$$

$$S = S_1 \text{ xor } C_{in}$$

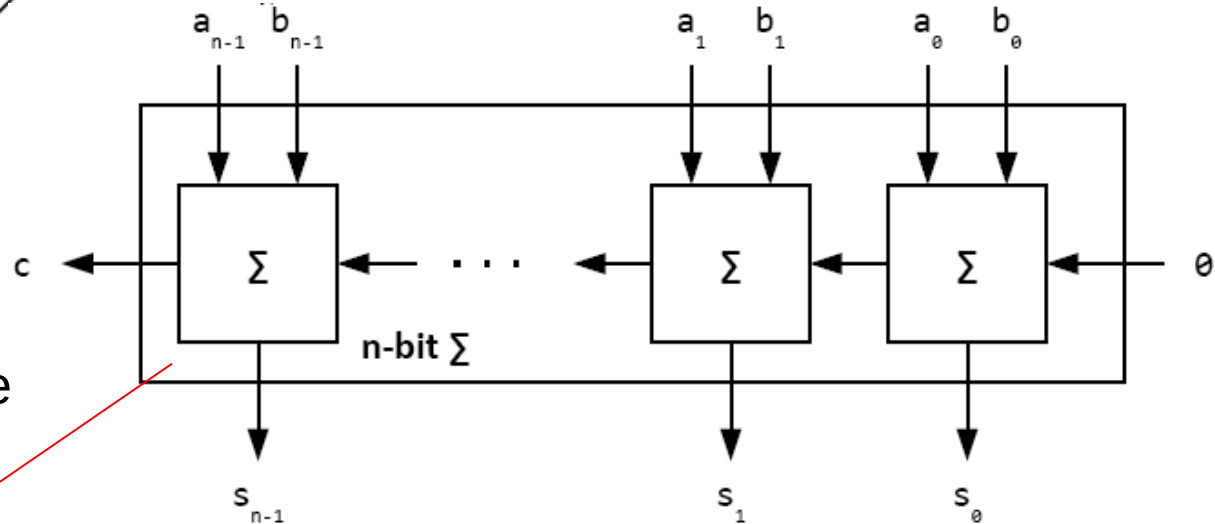
$$Cout_2 = S_1 \text{ and } C_{in}$$

$$Cout = Cout_1 \text{ or } Cout_2$$

Hardware of Ripple-Carry Adder

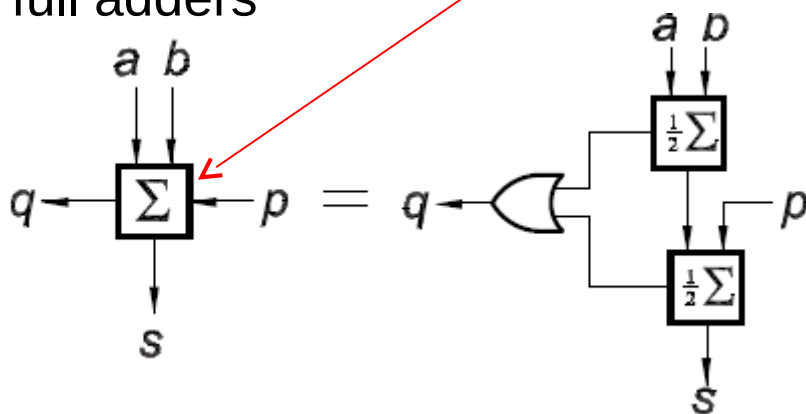


Common symbol for adder

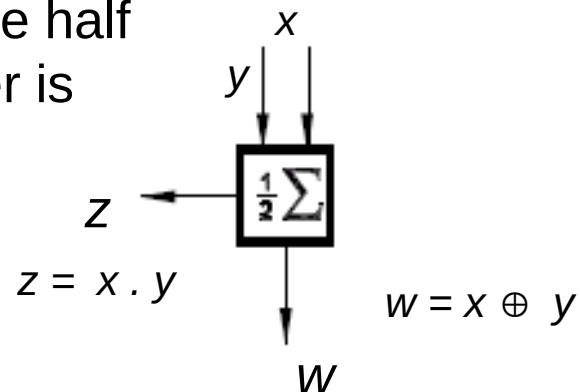


Internal structure

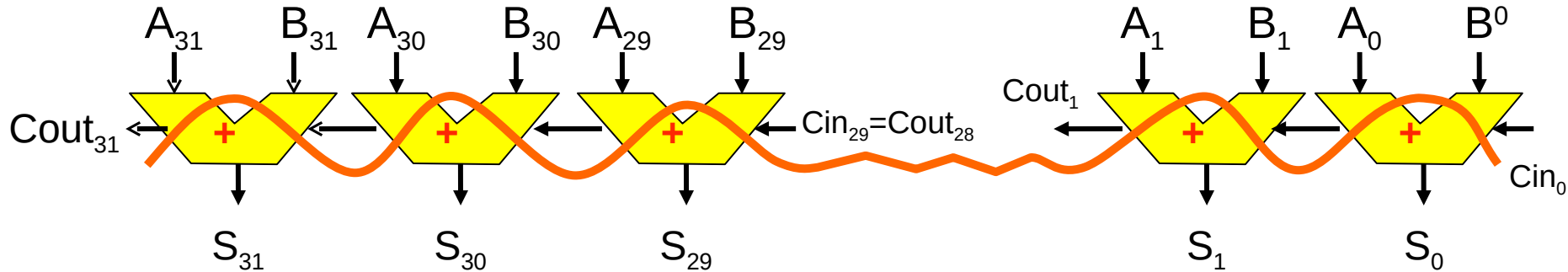
Realized by 1-bit full adders



where half adder is



Simple Adder



Simplest N-bit adder

we chain 1-bit full adders

"Carry" ripple through their chain

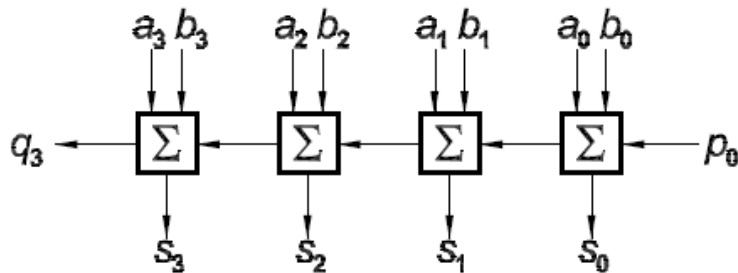
Minimal number of logical elements

Delay is given by the last Cout - $2 \cdot (N-1) + 3$ gates of the last adder

= $(2N+1)$ times propagation delay of 1 gate

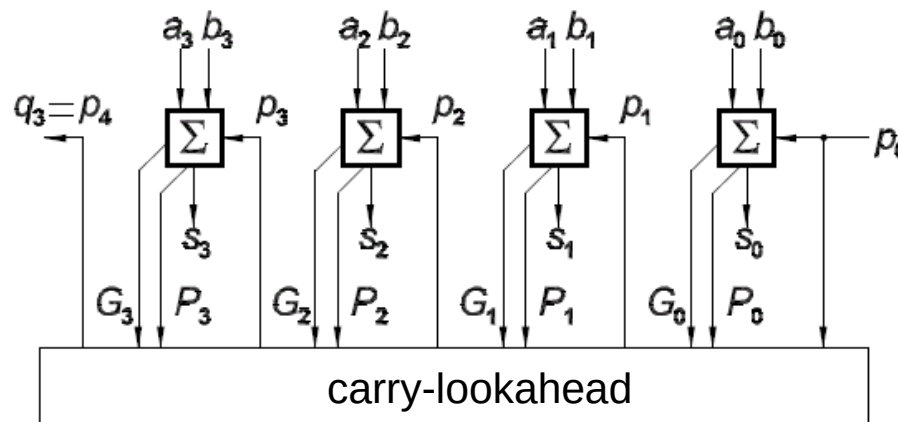
Speed of the Adder

- Parallel adder is combinational logic/circuit. Is there any reason to speak about its speed? Try to describe!
- Yes, and it is really slow. Why?
- Possible enhancement – adder with carry-lookahead (CLA) logic!



CLA – Carry-lookahead

- Adder combined with CLA provides enough speedup when compared with parallel ripple-carry adder and yet number of additional gates is acceptable
- CLA for 64-bit adder increases hardware price for about 50% but the speed is increased (signal propagation time decreased) 9 times.
- The result is significant speed/price ratio enhancement.



The Basic Equations for the CLA Logic

- the **G**eneration of carry on position (bit) j is defined as:

$$g_j = x_j y_j$$

- the need for carry **P**ropagation from previous bit:

$$p_j = x_j \oplus y_j = x_j \bar{y}_j \vee \bar{x}_j y_j$$

- Then:

- the result of sum for bit j is given by:

$$s_j = c_j (\overline{x_j \oplus y_j}) \vee \bar{c}_j (x_j \oplus y_j) = c_j \bar{p}_j \vee \bar{c}_j p_j = p_j \oplus c_j$$

- and carry to the higher order bit ($j+1$) is given by:

$$c_{j+1} = x_j y_j \vee (x_j \oplus y_j) c_j = g_j \vee p_j c_j$$

CLA

The carry can be computed as:

$$c_1 = g_0 \vee p_0 c_0$$

$$c_2 = g_1 \vee p_1 c_1 = g_1 \vee p_1 (g_0 \vee p_0 c_0) = g_1 \vee p_1 g_0 \vee p_1 p_0 c_0$$

$$c_3 = g_2 \vee p_2 c_2 = g_2 \vee p_2 (g_1 \vee p_1 g_0 \vee p_1 p_0 c_0) = g_2 \vee p_2 g_1 \vee p_2 p_1 g_0 \vee p_2 p_1 p_0 c_0$$

$$c_4 = g_3 \vee p_3 c_3 = g_3 \vee p_3 g_2 \vee p_3 p_2 g_1 \vee p_3 p_2 p_1 g_0 \vee p_3 p_2 p_1 p_0 c_0$$

$$c_5 = \dots$$

Description of the equation for c_3 as an example:

The carry input for bit 3 is active **when** carry is generated in bit 2 **or** carry propagates condition holds for bit 2 and carry is generated in the bit 1 **or** both bits 2 and 1 propagate carry and carry is generated in bit 0

Fast Parallel Adder Realization and Limits

- The previous, cascade based adder is slow – carry propagation delay
- The parallel adder is combinatorial circuit, it can be realized through sum of minterms (product of sums), two levels of gates (wide number of inputs required)
- But for 64-bit adder 10^{20} gates is required

Solution #1

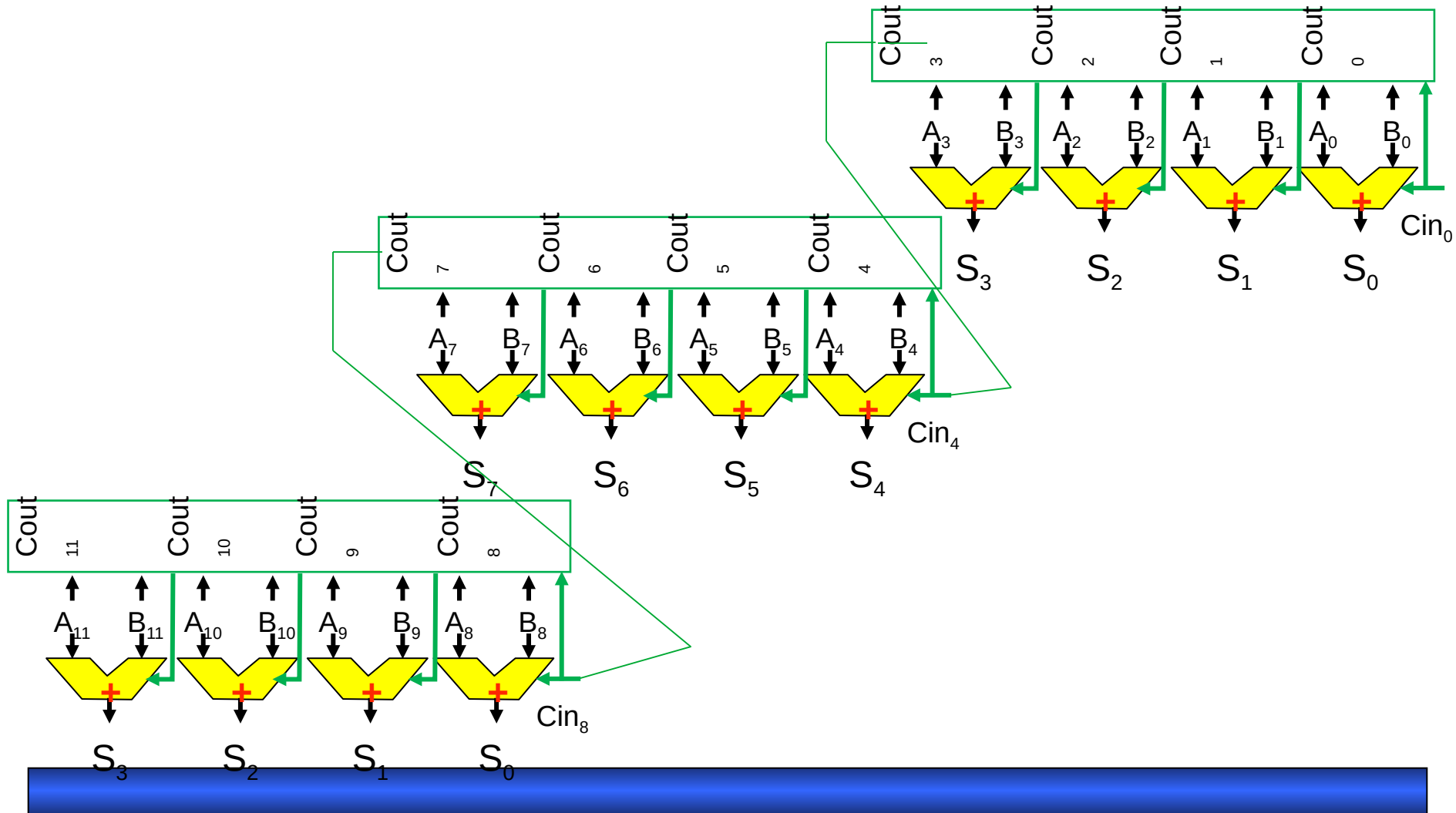
- Use of carry-lookahead circuits for small number of bits and cascade of adders with fraction of the required width

Solution #2

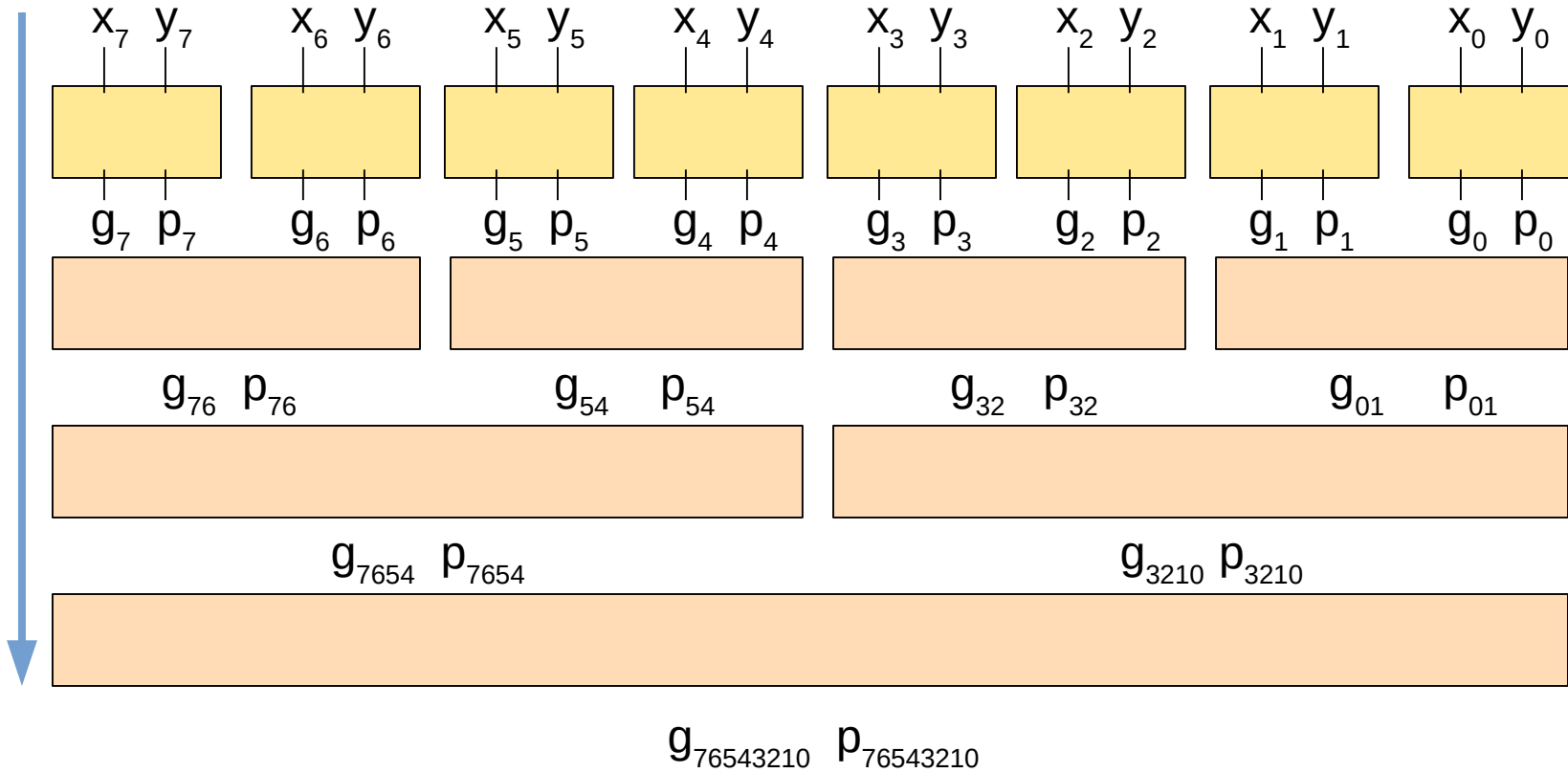
- Tree of CLA to speed up computation to $O(\log(n))$

32bit CLA "carry look-ahead" adder

In constant time compute 4-bits sum + carry



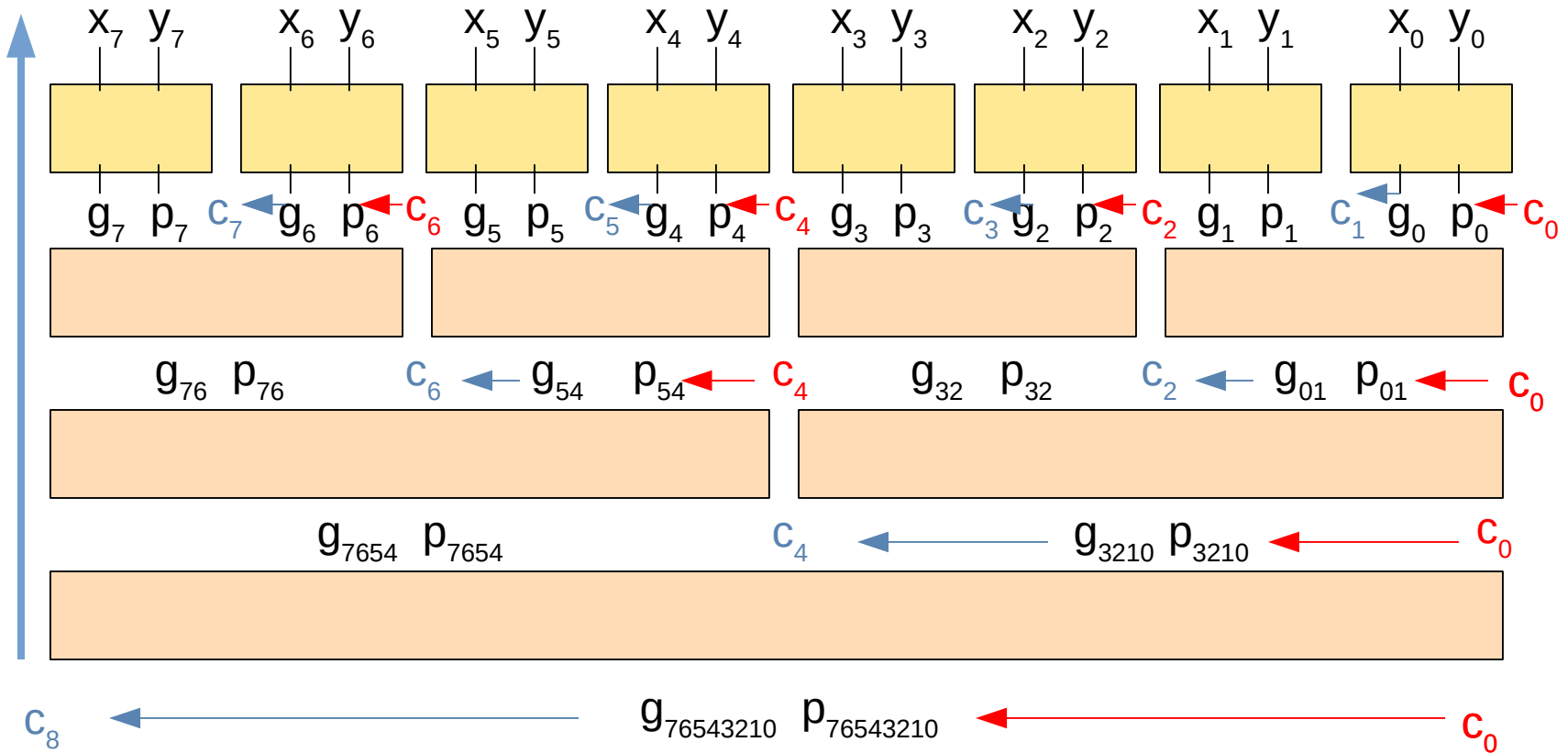
Tree of CLA



$$p_{out} = p_{hi} \text{ and } p_{lo}$$

$$g_{out} = g_{hi} \text{ or } (g_{lo} \text{ and } p_{hi})$$

Tree of CLA



$$c_{out} = g \text{ or } (p \text{ and } c_{in})$$

Increment / Decrement

Very fast operations that do not need an adder!

The last bit is always negated, and the previous ones are negated according to the end 1 / 0

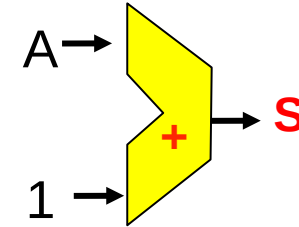
Dec.	Binary 8 4 2 1	+1	Binary 8 4 2 1	-1
0	0000	0001	0000	1111
1	0001	0010	0001	0000
2	0010	0011	0010	0001
3	0011	0100	0011	0010
4	0100	0101	0100	0011
5	0101	0110	0101	0100
6	0110	0111	0110	0101
7	0111	1000	0111	0110
8	1000	1001	1000	0111
9	1001	1010	1001	1000
10	1010	1011	1010	1001
11	1011	1100	1011	1010
12	1100	1101	1100	1011
13	1101	1110	1101	1100
14	1110	1111	1110	1101
15	1111	0000	1111	1110

Special Case Add +1/-1

$$S_0 = \text{not } A_0$$

$$S_1 = A_1 \text{ xor } A_0$$

$$S_2 = A_2 \text{ xor } (A_1 \text{ and } A_0)$$

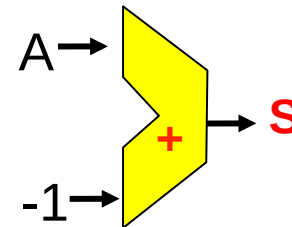


$$\text{Eq: } S_i = A_i \text{ xor } (A_{i-1} \text{ and } A_{i-2} \text{ and } \dots \text{ and } A_1 \text{ and } A_0); i=0..n-1$$

$$S_0 = \text{not } A_0$$

$$S_1 = A_1 \text{ xor } (\text{not } A_0)$$

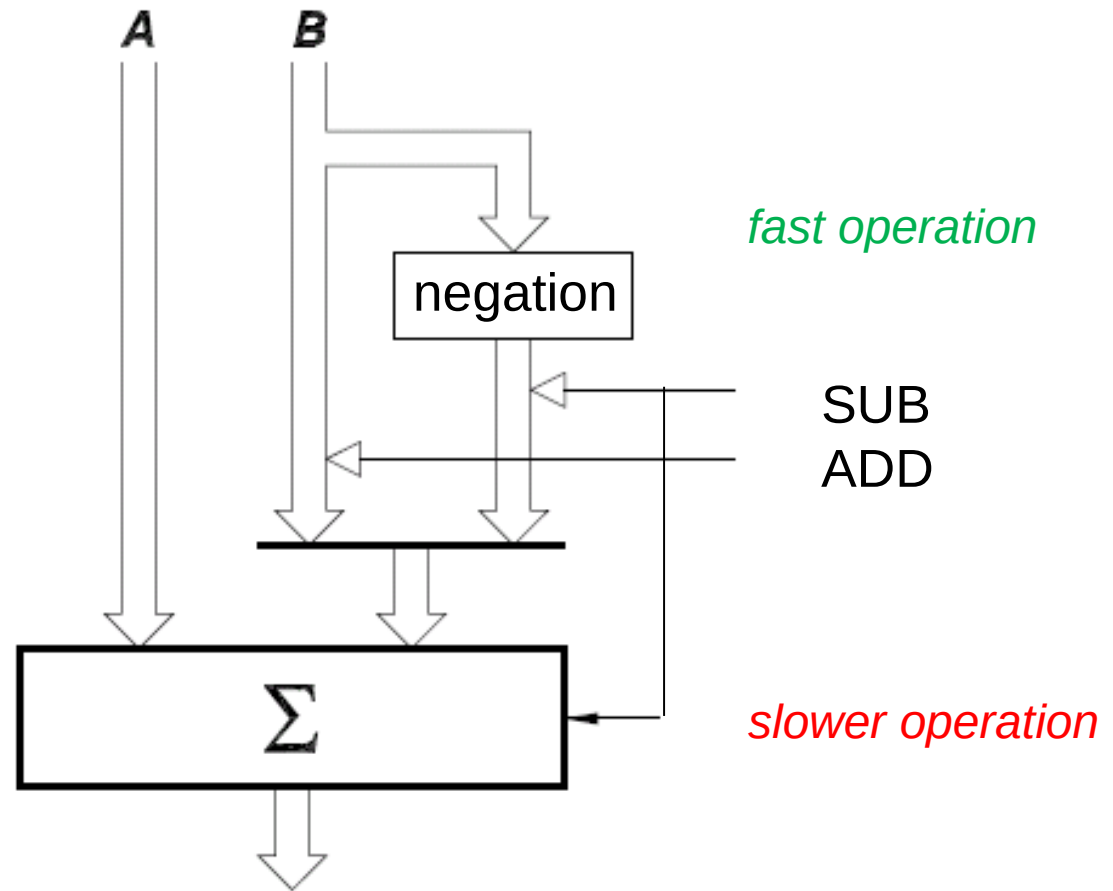
$$S_2 = A_2 \text{ xor } (\text{not } A_1 \text{ and } \text{not } A_0)$$



$$\text{Eq: } S_i = A_i \text{ xor } (\text{not } A_{i-1} \text{ and } \dots \text{ and } \text{not } A_0); i=0..n-1$$

The number of circuits is given by the arithmetic series, with the complexity $O(n^2)$ where n is the number of bits. The operation can be performed in parallel for all bits, and for the both +1/-1 operations, we use a circuit that differs only by negations.

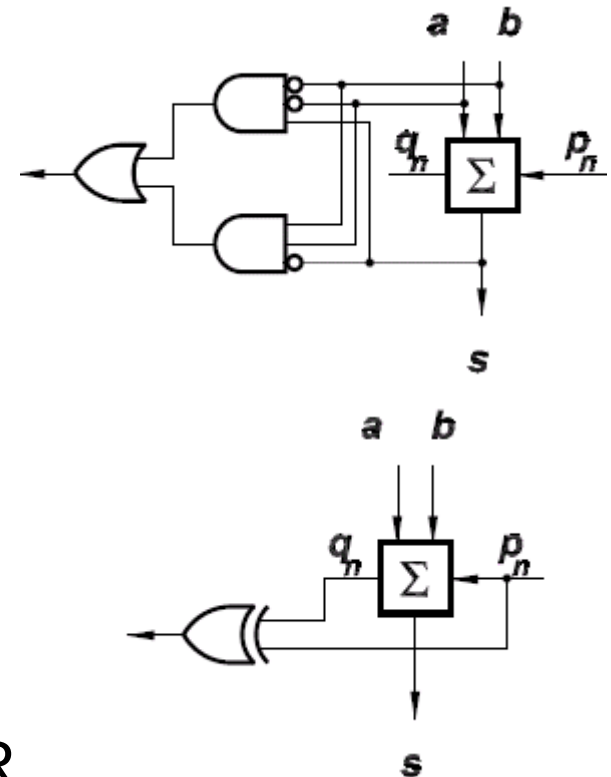
Addition / Subtraction HW



Source: X36JPO, A. Pluháček

Arithmetic Overflow (Underflow)

- Result of the arithmetic operation is incorrect because, it does not fit into selected number of the representation bits (width)
- But for the signed arithmetic, it is not equivalent to the carry from the most significant bit.
- The arithmetic overflow is signaled if result sign is different from operand signs if both operands have same sign
- or can be detected with exclusive-OR of carry to and from the most significant bit



Overflow of Unsigned Number Binary Representation

- The carry from MSB (the most significant bit) is observed in this case
- The arithmetic result is incorrect because it is out of range.

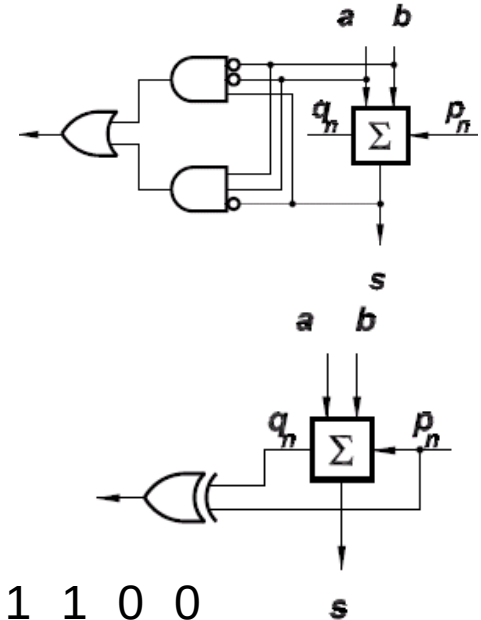
For 5 bit representation:

$ \begin{array}{r} 28 \quad 1\ 1\ 1\ 0\ 0 \\ +5 \quad +\ 0\ 0\ 1\ 0\ 1 \\ \hline ?1 \quad 1\ 0\ 0\ 0\ 0\ 1 \end{array} $ <p style="text-align: center;"> </p>	$ \begin{array}{r} 12 \quad 0\ 1\ 1\ 0\ 0 \\ +5 \quad +\ 0\ 0\ 1\ 0\ 1 \\ \hline 17 \quad 0\ 1\ 0\ 0\ 0\ 1 \end{array} $ <p style="text-align: center;"> </p>	$ \begin{array}{r} 28 \quad 1\ 1\ 1\ 0\ 0 \\ 21 \quad +\ 1\ 0\ 1\ 0\ 1 \\ \hline ?17 \quad 1\ 1\ 0\ 0\ 0\ 1 \end{array} $ <p style="text-align: center;"> </p>
$ \begin{array}{r} 28 \quad 1\ 1\ 1\ 0\ 0 \\ +5 \quad +\ 0\ 0\ 1\ 0\ 1 \\ \hline ?1 \quad 1\ 0\ 0\ 0\ 0\ 1 \end{array} $ <p style="text-align: center;"> </p>	$ \begin{array}{r} 28 \quad 1\ 1\ 1\ 0\ 0 \\ +5 \quad +\ 0\ 0\ 1\ 0\ 1 \\ \hline 19 \quad 1\ 0\ 0\ 1\ 1 \end{array} $ <p style="text-align: center;"> </p>	$ \begin{array}{r} 28 \quad 1\ 1\ 1\ 0\ 0 \\ 21 \quad +\ 1\ 0\ 1\ 0\ 1 \\ \hline ?17 \quad 1\ 1\ 0\ 0\ 0\ 1 \end{array} $ <p style="text-align: center;"> </p>

The incorrect result is smaller than each of addends

Overflow of Signed Binary Representation

- Result is incorrect, numeric value is outside of the range that can be represented with a given number of digits
- **It is manifested by result sign different from the sign of addends when both addends signs are the same, and**
- the exclusive-or (xor) of carry **to** and **from** MSB differs.



For 5 bit representation:

<table border="0" style="width: 100%;"> <tr> <td style="background-color: yellow; padding: 5px;">-4</td> <td style="padding: 5px;">1 1 1 0 0</td> <td></td> </tr> <tr> <td style="padding: 5px;">+5</td> <td style="padding: 5px;">+ 0 0 1 0 1</td> <td style="padding: 5px;">12</td> </tr> <tr> <td style="border-top: 1px solid black; padding: 5px;">1</td> <td style="border-top: 1px solid black; padding: 5px;">1 0 0 0 0 1</td> <td style="padding: 5px;">+5</td> </tr> </table> <p style="text-align: right; margin-right: 20px;">✓</p>	-4	1 1 1 0 0		+5	+ 0 0 1 0 1	12	1	1 0 0 0 0 1	+5	<table border="0" style="width: 100%;"> <tr> <td style="background-color: yellow; padding: 5px;">-4</td> <td style="padding: 5px;">1 1 1 0 0</td> <td></td> </tr> <tr> <td style="background-color: yellow; padding: 5px;">-11</td> <td style="padding: 5px;">+ 1 0 1 0 1</td> <td style="padding: 5px;">0 1 1 0 0</td> </tr> <tr> <td style="border-top: 1px solid black; padding: 5px;">-15</td> <td style="border-top: 1px solid black; padding: 5px;">1 1 0 0 0 1</td> <td style="padding: 5px;">+5</td> </tr> </table> <p style="text-align: right; margin-right: 20px;">✓</p>	-4	1 1 1 0 0		-11	+ 1 0 1 0 1	0 1 1 0 0	-15	1 1 0 0 0 1	+5	<table border="0" style="width: 100%;"> <tr> <td style="padding: 5px;">0 1 1 0 0</td> <td></td> </tr> <tr> <td style="padding: 5px;">+ 0 0 1 0 1</td> <td style="padding: 5px;">12</td> </tr> <tr> <td style="border-top: 1px solid black; padding: 5px;">1 0 0 0 0 1</td> <td style="padding: 5px;">+5</td> </tr> </table> <p style="text-align: right; margin-right: 20px;">✗</p>	0 1 1 0 0		+ 0 0 1 0 1	12	1 0 0 0 0 1	+5	<table border="0" style="width: 100%;"> <tr> <td style="background-color: yellow; padding: 5px;">-4</td> <td style="padding: 5px;">1 1 1 0 0</td> <td></td> </tr> <tr> <td style="background-color: yellow; padding: 5px;">-13</td> <td style="padding: 5px;">+ 1 0 0 1 1</td> <td style="padding: 5px;">0 1 1 0 0</td> </tr> <tr> <td style="border-top: 1px solid black; padding: 5px;">?15</td> <td style="border-top: 1px solid black; padding: 5px;">1 0 1 1 1 1</td> <td style="padding: 5px;">+5</td> </tr> </table> <p style="text-align: right; margin-right: 20px;">✗</p>	-4	1 1 1 0 0		-13	+ 1 0 0 1 1	0 1 1 0 0	?15	1 0 1 1 1 1	+5
-4	1 1 1 0 0																																			
+5	+ 0 0 1 0 1	12																																		
1	1 0 0 0 0 1	+5																																		
-4	1 1 1 0 0																																			
-11	+ 1 0 1 0 1	0 1 1 0 0																																		
-15	1 1 0 0 0 1	+5																																		
0 1 1 0 0																																				
+ 0 0 1 0 1	12																																			
1 0 0 0 0 1	+5																																			
-4	1 1 1 0 0																																			
-13	+ 1 0 0 1 1	0 1 1 0 0																																		
?15	1 0 1 1 1 1	+5																																		

Sign Extension

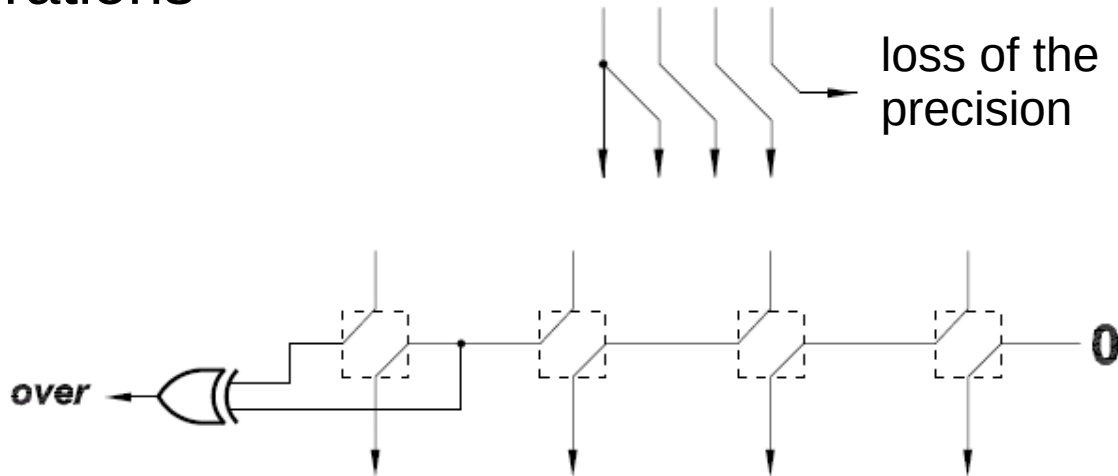
Example:

```
short int x = 15213;  
int      ix = (int) x;  
short int y = -15213;  
int      iy = (int) y;
```

	Decimal	Hex	Binary
x	15213	3B 6D	00111011 01101101
ix	15213	00 00 C4 92	00000000 00000000 00111011 01101101
y	-15213	C4 93	11000100 10010011
iy	-15213	FF FF C4 93	11111111 11111111 11000100 10010011

Arithmetic Shift to the Left and to the Right

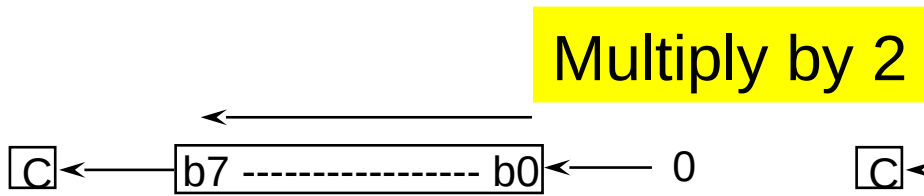
- arithmetic shift by one to the left/right is equivalent to signed multiply/divide by 2 (digits movement in positional (place-value) representation)
- Notice difference between arithmetic, logic and cyclic shift operations



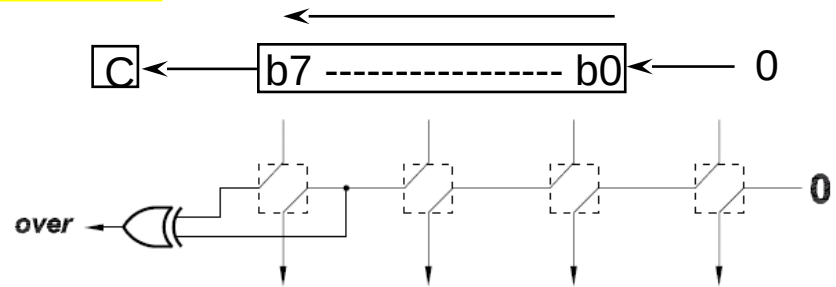
- Remark: Barrel shifter can be used for fast variable shifts

Multiply/Divide by 2

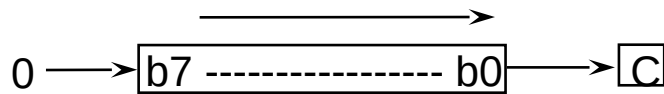
Logical Shift



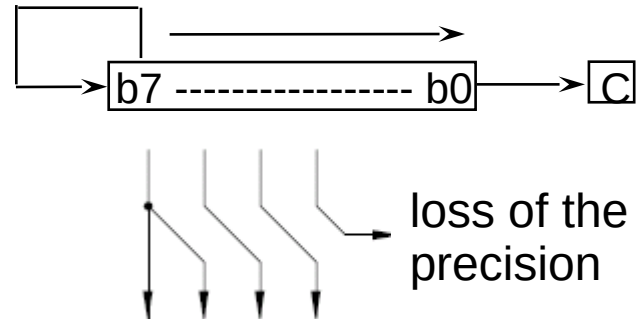
Arithmetic Shift



Divide by 2 unsigned

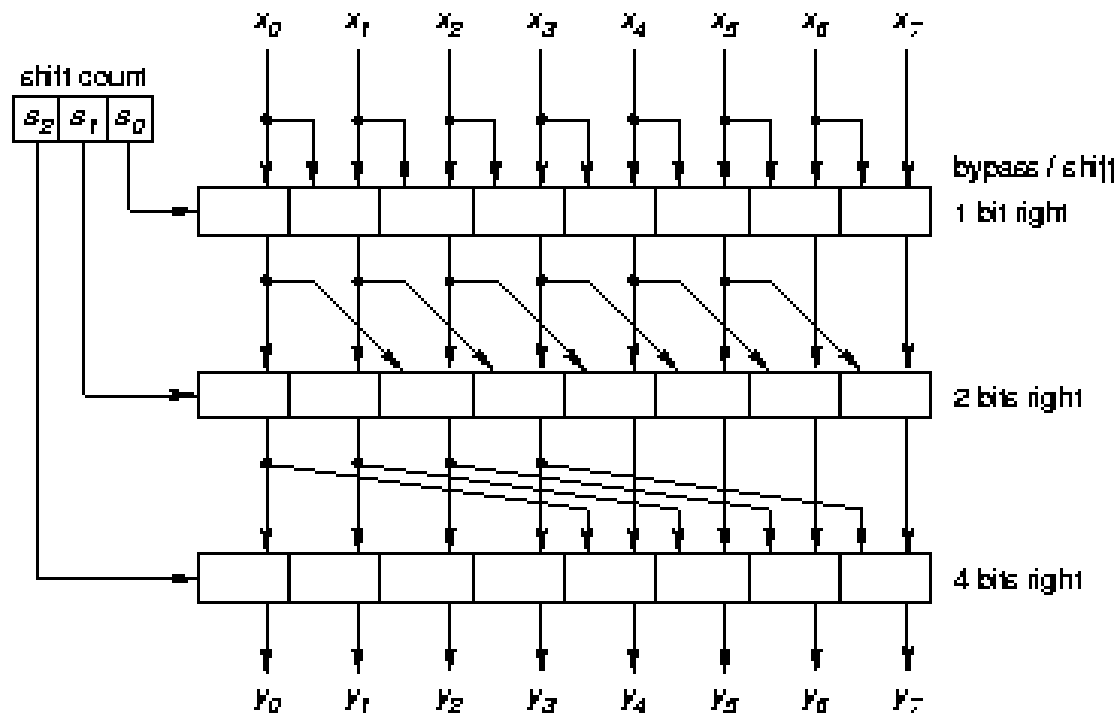


Divide by 2 signed



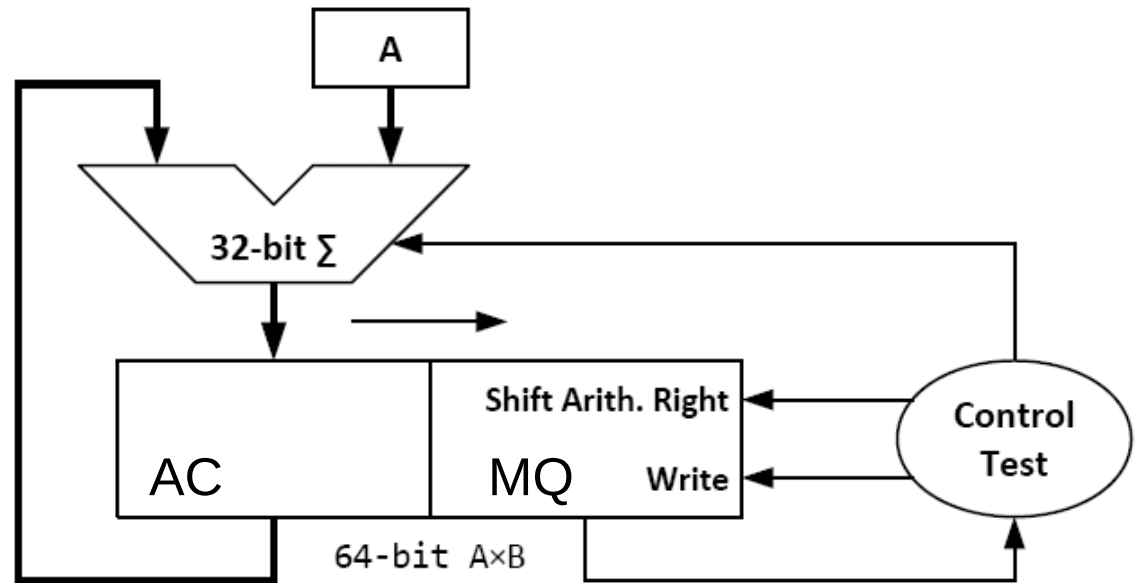
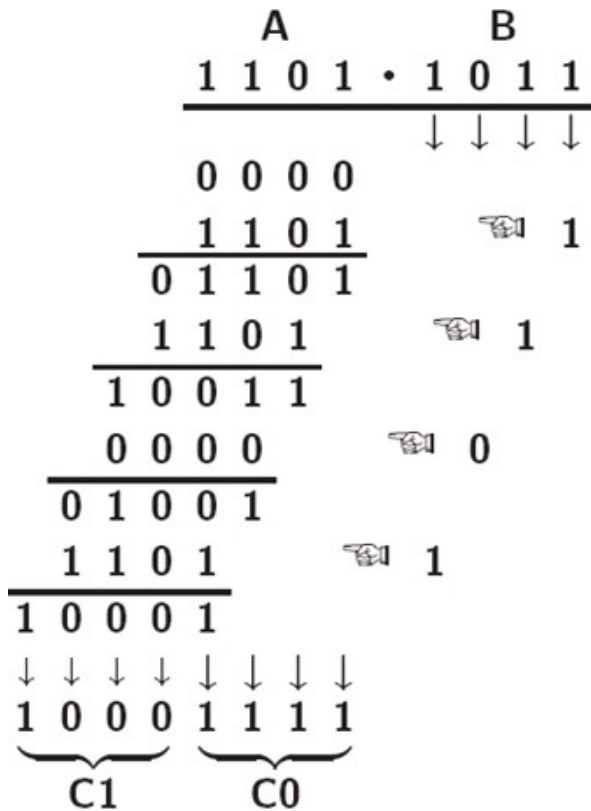
C represents Carry Flag, it is present only on some processors: x86/ARM yes, MIPS no

Barrel Shifter



Barrel shifter can be used for fast variable shifts

Sequential Hardware Multiplier (32b Case)



The speed of the multiplier is horrible

Algorithm for Multiplication

A = multiplicand;

MQ = multiplier;

AC = 0;

for(int i=0; i < n; i++) // n – represents number of bits

{

if(MQ₀ == 1) AC = AC + A; // MQ₀ = LSB of MQ

SR (shift AC MQ by one bit right and insert information about
carry from the MSB from previous step)

}

end.

when loop ends AC MQ holds 64-bit result

Example of the Multiply X by Y

Multiplicand $x=110$ and multiplier $y=101$. $x \rightarrow A$, $y \rightarrow MQ$

i	operation	C	AC	MQ	A	comment
		0	000	101	110	initial setup
1	AC = AC+A	0	110	101		start of the cycle
	SR	0	011	010		
2	nothing	0	011	010		because of $MQ_0 = 0$
	SR	0	001	101		
3	AC = AC+A	0	111	101		
	SR	0	011	110		end of the cycle

The whole operation: $x \times y = 110 \times 101 = 011110$, ($6 \times 5 = 30$)

Carry, AC can be mapped to AC, MQ.MSB and A masked by MQ.LSB, then if, add and shift steps are combined to the one.

Signed Multiplication by Unsigned HW for two's Complement

One possible solution (for illustration only)

$$C = A \cdot B$$

Let A and B representations are n bits and result is 2n bits

$$D(C) = D(A) \cdot D(B)$$

$$- (D(B) \ll n) \quad \text{if } A < 0$$

$$- (D(A) \ll n) \quad \text{if } B < 0$$

Consider for negative numbers

$$(2^n + A) \cdot (2^n + B) = 2^{2n} + 2^n A + 2^n B + A \cdot B$$

where 2^{2n} is out of the result representation, next two elements have to be eliminated if input is negative

Today more often used separate sign evaluation and multiplication of absolute values.

Wallace Tree Based Multiplier – Required Function

$Q=X.Y$, X and Y are considered as and 8bit unsigned numbers

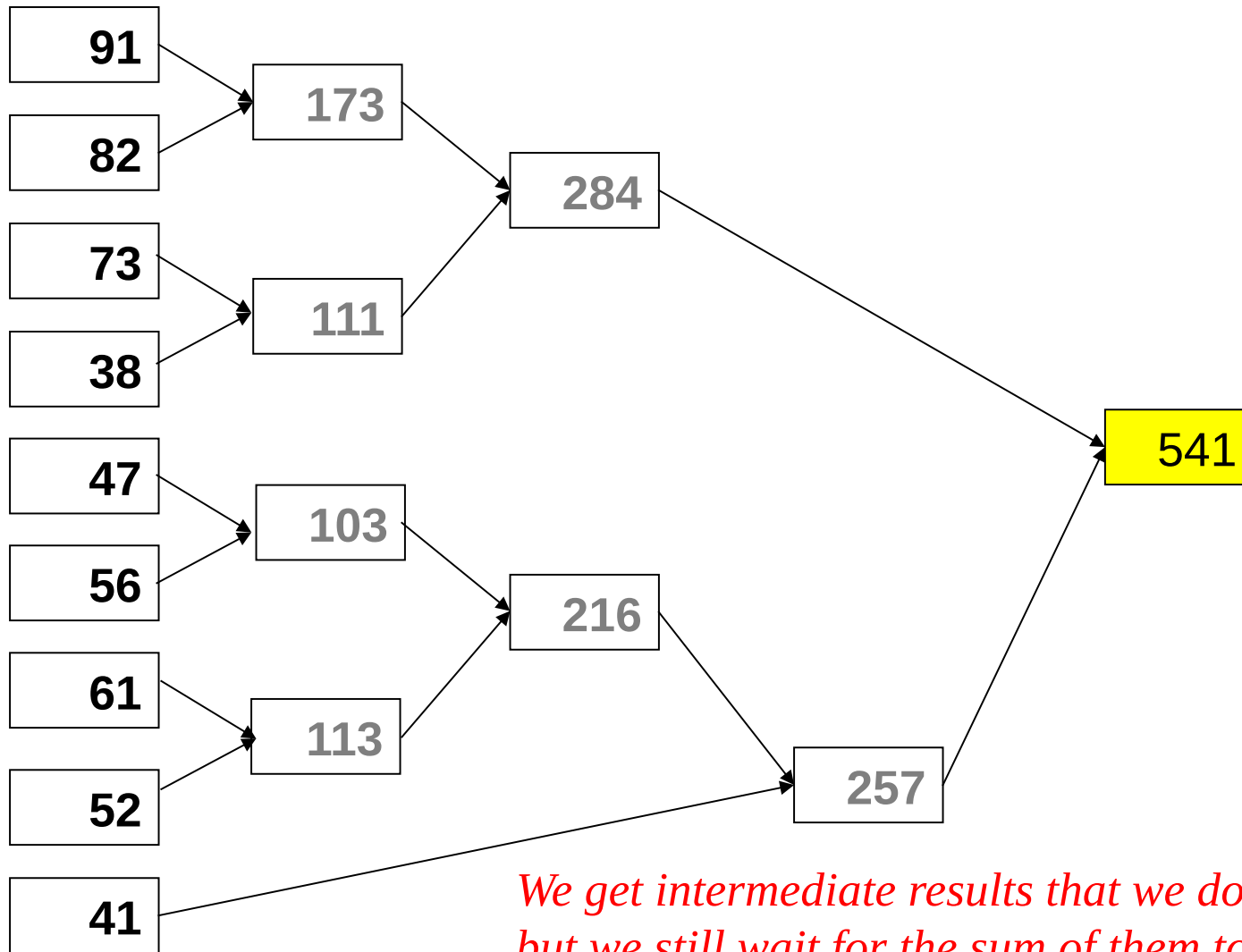
$$(x_7 x_6 x_5 x_4 x_3 x_2 x_1 x_0) \cdot (y_7 y_6 y_5 y_4 y_3 y_2 y_1 y_0) =$$

0	0	0	0	0	0	0	0	x_7y_0	x_6y_0	x_5y_0	x_4y_0	x_3y_0	x_2y_0	x_1y_0	x_0y_0	P0
0	0	0	0	0	0	0	x_7y_1	x_6y_1	x_5y_1	x_4y_1	x_3y_1	x_2y_1	x_1y_1	x_0y_1	0	P1
0	0	0	0	0	0	x_7y_2	x_6y_2	x_5y_2	x_4y_2	x_3y_2	x_2y_2	x_1y_2	x_0y_2	0	0	P2
0	0	0	0	0	x_7y_3	x_6y_3	x_5y_3	x_4y_3	x_3y_3	x_2y_3	x_1y_3	x_0y_3	0	0	0	P3
0	0	0	0	x_7y_4	x_6y_4	x_5y_4	x_4y_4	x_3y_4	x_2y_4	x_1y_4	x_0y_4	0	0	0	0	P4
0	0	0	x_7y_5	x_6y_5	x_5y_5	x_4y_5	x_3y_5	x_2y_5	x_1y_5	x_0y_5	0	0	0	0	0	P5
0	0	x_7y_6	x_6y_6	x_5y_6	x_4y_6	x_3y_6	x_2y_6	x_1y_6	x_0y_6	0	0	0	0	0	0	P6
0	x_7y_7	x_6y_7	x_5y_7	x_4y_7	x_3y_7	x_2y_7	x_1y_7	x_0y_7	0	0	0	0	0	0	0	P7
Q_{15}	Q_{14}	Q_{13}	Q_{12}	Q_{11}	Q_{10}	Q_9	Q_8	Q_7	Q_6	Q_5	Q_4	Q_3	Q_2	Q_1	Q_0	

The sum of $P0+P1+\dots+P7$ gives result of X and Y multiplication.

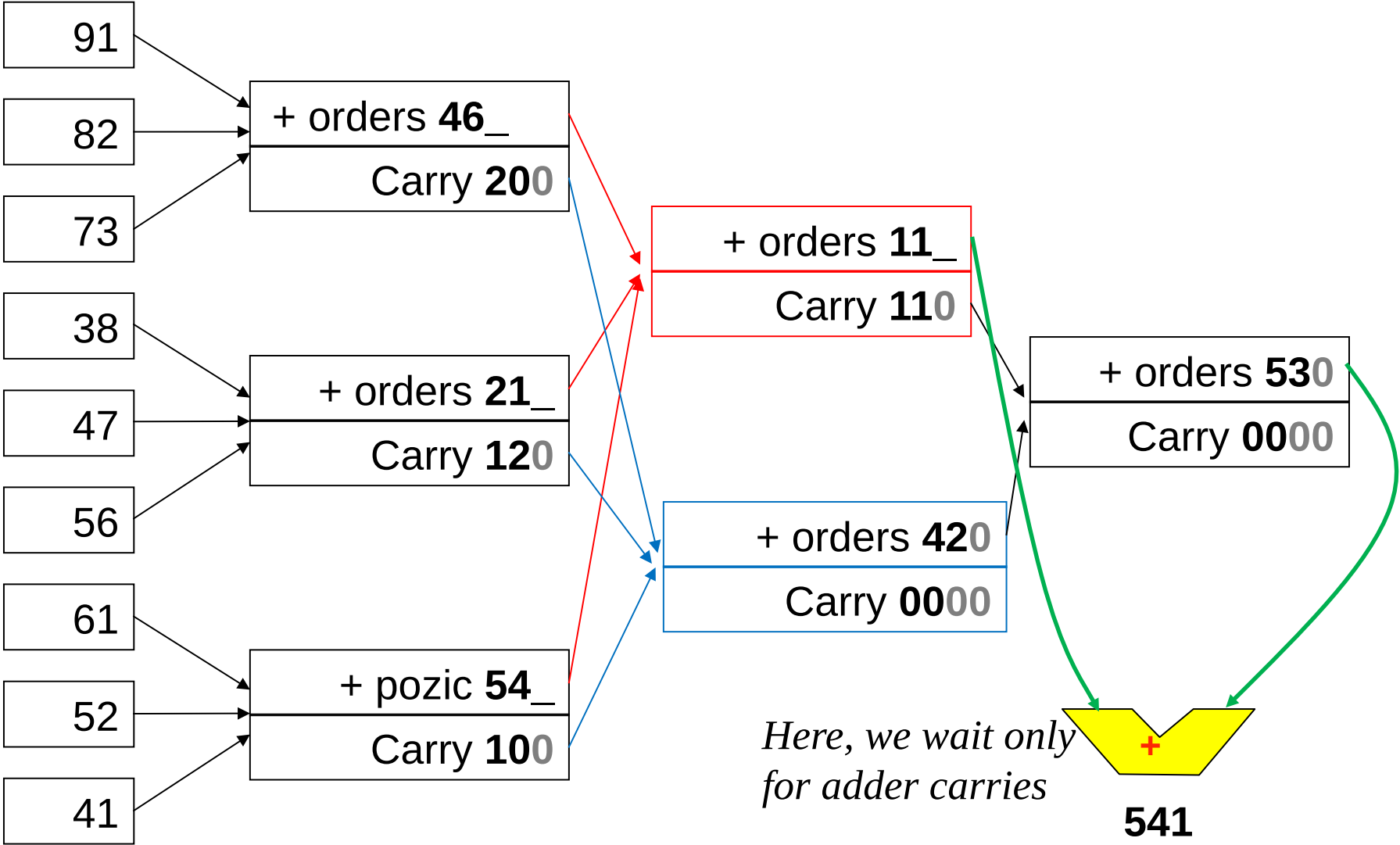
$$Q = X.Y = P0 + P1 + \dots + P7$$

Idea to Consider – Parallel Adder of 9 Decimal Numbers



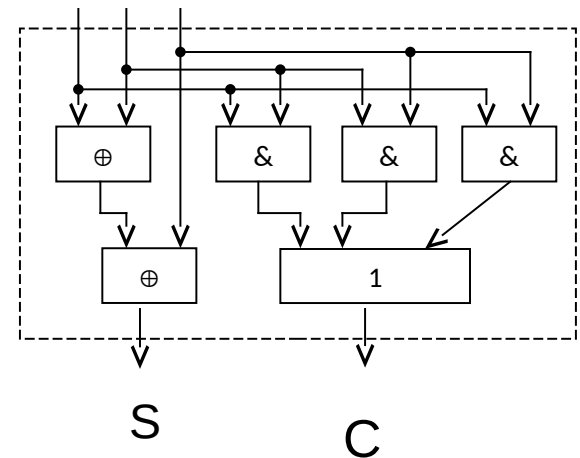
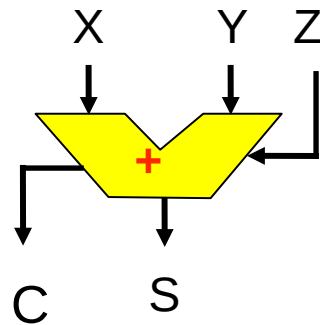
We get intermediate results that we do not need at all, but we still wait for the sum of them to finish!

Decadic Carry-save Adder

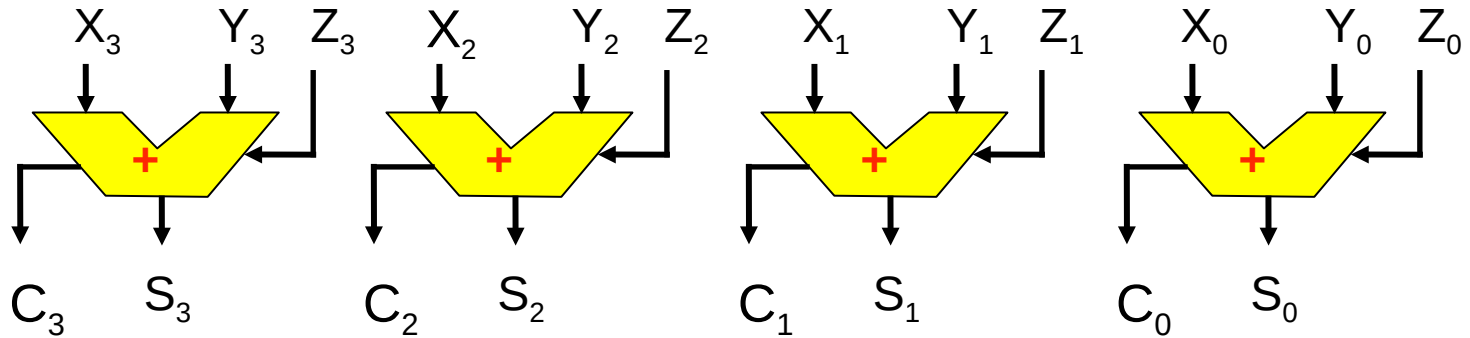


1bit Carry Save Adder

X	0	0	1	1	0	0	1	1
Y	0	1	0	1	0	1	0	1
Z	0	0	0	0	1	1	1	1
Sum	0	1	1	0	1	0	0	1
C=Cout	0	0	0	1	0	1	1	1



4bit Carry Save Adder



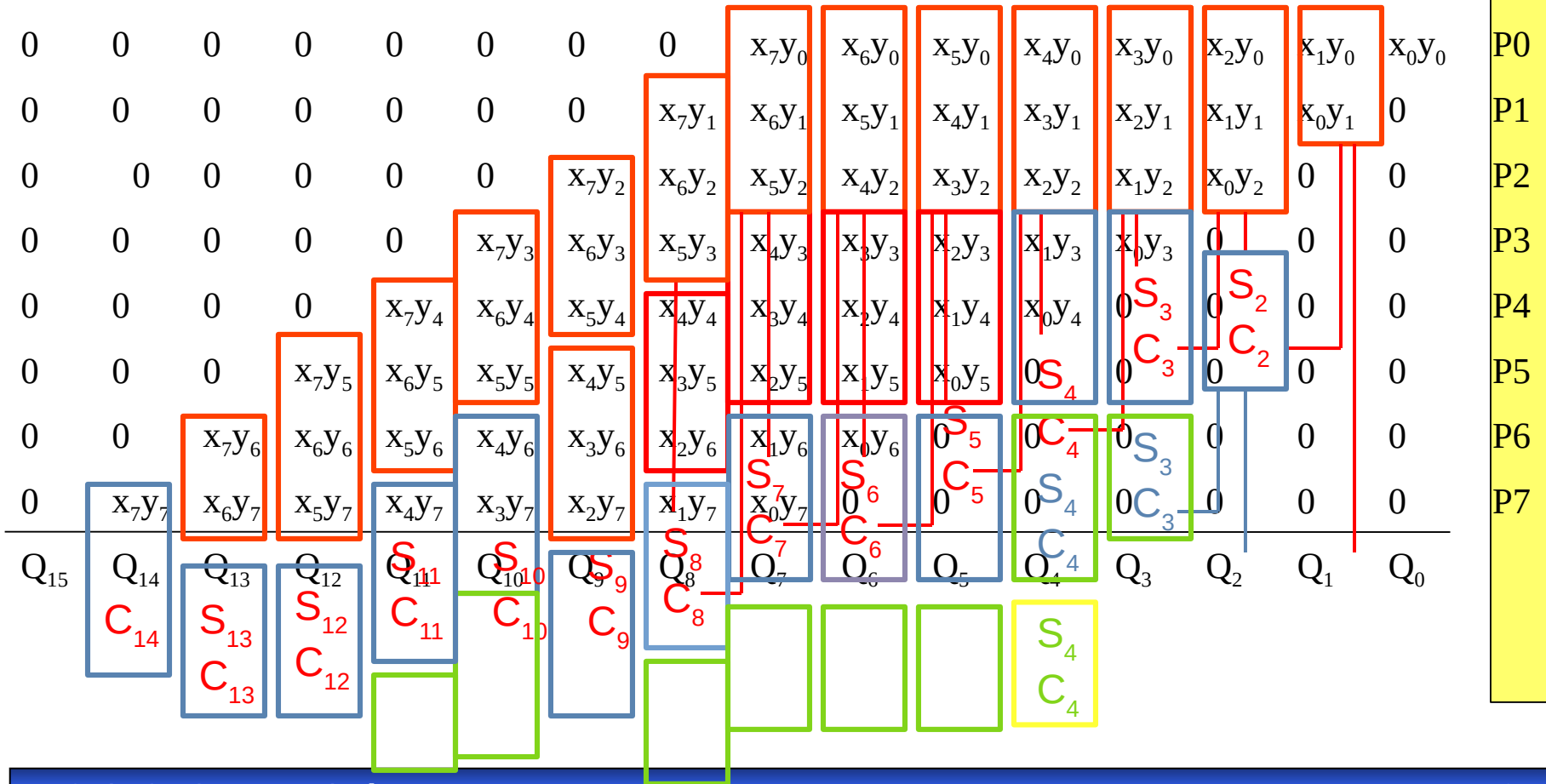
$$\begin{array}{r}
 X_3 X_2 X_1 X_0 \\
 Y_3 Y_2 Y_1 Y_0 \\
 \hline
 Z_3 Z_2 Z_1 Z_0 \\
 \hline
 S_3 S_2 S_1 S_0 \\
 C_3 C_2 C_1 C_0
 \end{array}$$

To get result we need to add $C+S$ by ripple adder or some fast adder

Wallace Tree Based Multiplier – CSA

$Q = X \cdot Y$, X and Y are considered as 8bit unsigned numbers

$$(x_7 x_6 x_5 x_4 x_3 x_2 x_1 x_0) \cdot (y_7 y_6 y_5 y_4 y_3 y_2 y_1 y_0) =$$



Speed of Arithmetic Operations

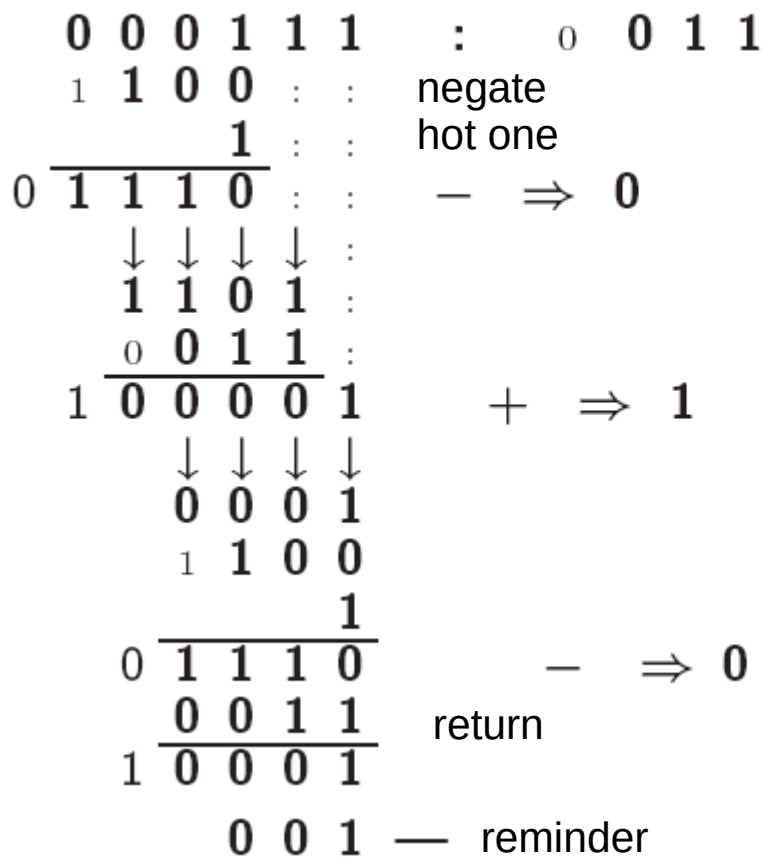
Operation	C language operator
Bitwise complement (negation)	$\sim x$
Multiply and divide by 2^n	$x \ll n$, $x \gg n$
Increment, decrement	$++x$, $x++$, $--x$, $x--$
Negate ← complement + increment	$-x$
Addition	$x+y$
Subtraction ← negation + addition	$x-y$
Multiply on hardware multiplier	$x*y$
Multiply on sequential multiplier/SW	
Division	x/y

Hardware Divider – Simple Sequential Algorithm

Non-restoring division

$$\boxed{111 : 011}$$

$7 / 3$
 $7 - 4 * 3 = -5$
 (non-restoring)
 $-5 + 2 * 3 = 1$
 $= 7 - 2 * 3$
 $1 - 3 = -2$
 (restoring)
 $-2 + 3 = 1$
Restoring is required only for last operation

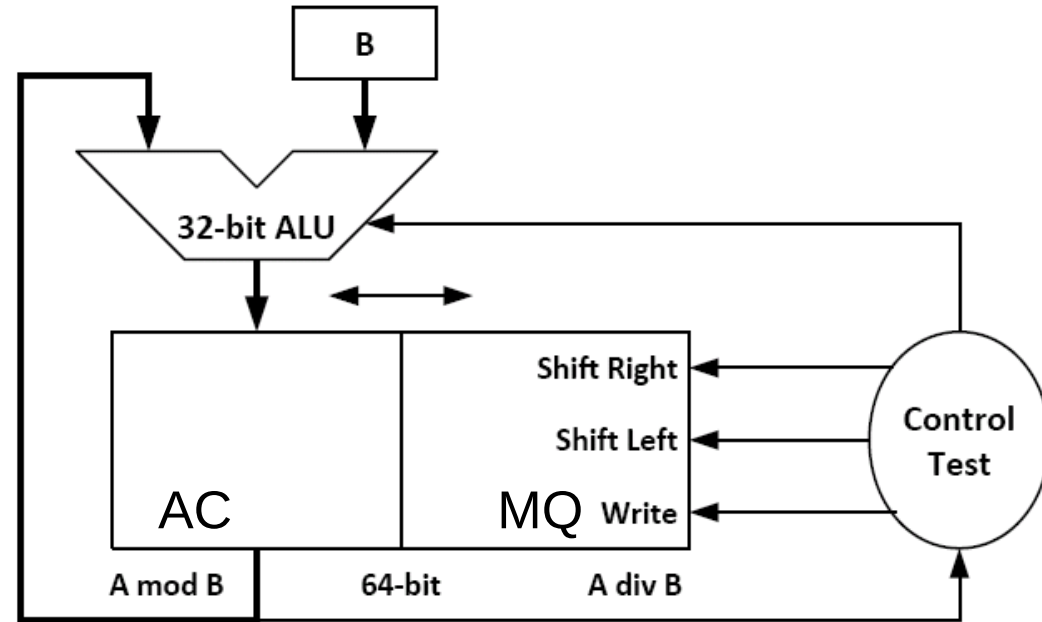


ALU does not check, if the dividend is smaller or not than divisor. It finds that during subtraction and needs to correct the result by addition.

Hardware divider logic (32b case)

$111 : 011$ dividend = quotient \times divisor + remainder

⊖	0 0 0 1 1 1	:	0 0 1 1	
	1 1 0 0	:	:	negate
	1	:	:	hot one
	0 1 1 1 0	:	:	- \Rightarrow 0
	↓ ↓ ↓ ↓	:	:	
	1 1 0 1	:	:	
⊕	0 0 1 1	:	:	
	1 0 0 0 0 1	:	:	+ \Rightarrow 1
	↓ ↓ ↓ ↓	:	:	
	0 0 0 1	:	:	
⊖	1 1 0 0	:	:	
	1	:	:	
	0 1 1 1 0	:	:	- \Rightarrow 0
	↓ ↓ ↓ ↓	:	:	
	0 0 1 1	:	:	return
⊖	1 0 0 0 1	:	:	
	0 0 1	-	-	remainder
	0 1 0	-	-	quotient



Algorithm of the sequential division

MQ = dividend;

B = divisor; (Condition: divisor is not 0!)

AC = 0;

```
for( int i=1; i <= n; i++)      {
    SL (shift AC MQ by one bit to the left, the LSB bit is kept on zero)
    if(AC >= B) {
        AC = AC - B;
        MQ0 = 1;      // the LSB of the MQ register is set to 1
    }
}
```

→ Value of MQ register represents quotient and AC remainder

Example of X/Y division

Dividend $x=1010$ and divisor $y=0011$

i	operation	AC	MQ	B	comment
		0000	1010	0011	initial setup
1	SL	0001	0100		
	nothing	0001	0100		the if condition not true
2	SL	0010	1000		
		0010	1000		the if condition not true
3	SL	0101	0000		$r \geq y$
	AC = AC - B; MQ₀ = 1;	0010	0001		
4	SL	0100	0010		$r \geq y$
	AC = AC - B; MQ₀ = 1;	0001	0011		end of the cycle

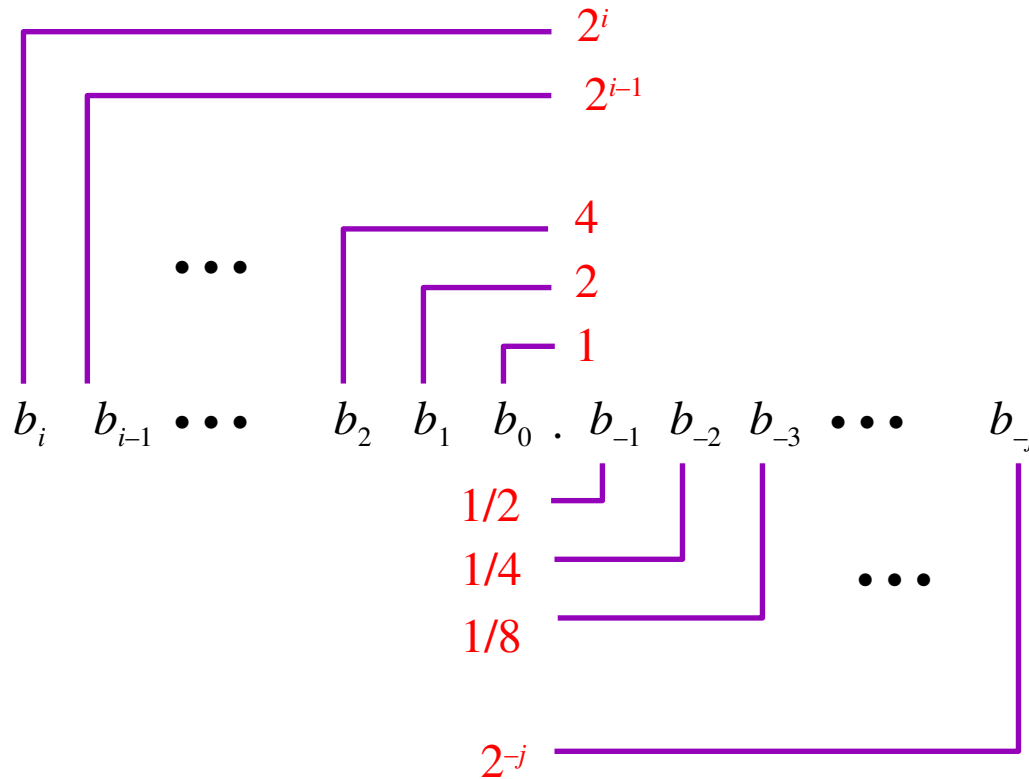
$x : y = 1010 : 0011 = 0011$ remainder 0001 , ($10 : 3 = 3$ remainder 1)

*Real Numbers

and their representation in computer

Real numbers

They can be used directly or as base for mantissa of float



$$\sum_{k=-j}^i b_k \cdot 2^k$$

Fixed Point Arithmetics - FX

- Similar to biased representation of negative numbers, the real number is represented by integer number:

$$A(X) = X \cdot K \qquad D(A) = \frac{A}{K}$$

- K is usually power of 2
- Addition and subtraction without problem
- Multiplication and division has problem as:

$$A(X \cdot Y) = (X \cdot Y) \cdot K = \frac{(X \cdot K) \cdot (Y \cdot K)}{K} = \frac{A(X) \cdot A(Y)}{K}$$

- If you know the range of numbers, e.g. $R \in [0; 1)$
 - $K = 2^n = 2^{32}$
 - You can get better precision than FPU
- Some SIMD are using fixed point numbers

Scientific Notation and Binary Numbers

Decimal number:

$$-123\ 000\ 000\ 000\ 000 \rightarrow -1.23 \times 10^{14}$$

$$0.000\ 000\ 000\ 000\ 000\ 123 \rightarrow +1.23 \times 10^{-16}$$

Binary number:

$$110\ 1100\ 0000\ 0000 \rightarrow 1.1011 \times 2^{14} = 29696_{10}$$

$$-0.0000\ 0000\ 0000\ 0001\ 1101 \rightarrow -1.1101 \times 2^{-16}$$

$$=-2.765655517578125 \times 10^{-5}$$

Higher Dynamic Range for Numbers (REAL/float)

- Scientific notation, semi-logarithmic, floating point
 - The value is represented by:
 - EXPONENT (E) – represents scale for given value
 - MANTISSA (M) – represents value in that scale
 - the sign(s) are usually separated as well
 - Mantissa x base^{Exponent}
- Normalized notation
 - The exponent and mantissa are adjusted such way, that mantissa is held in some standard range. Usually $\langle 0.5, \text{base} \rangle$
 - When considered base $z=2$ is considered then mantissa range is $\langle 1, 2 \rangle$ or alternatively $\langle 0.5, 1 \rangle$.
- Decimal representation: 7.26478×10^3
- Binary representation: $1,010011 \times 2^{1001}$

Standardized Format for REAL Type Numbers

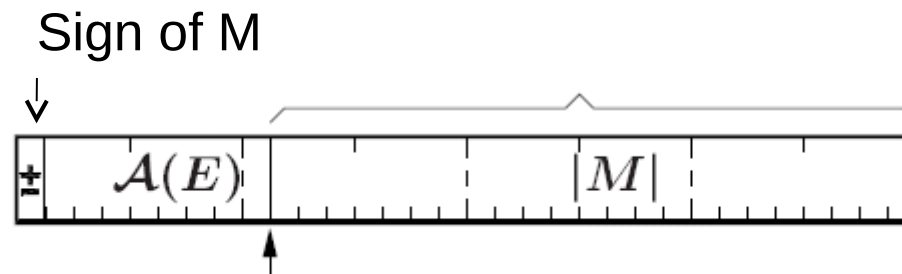
- Standard IEEE-754 defines next REAL representation and precision
 - single-precision – in the C language declared as **float**
 - uses 32 bits (1 + 8 + 23) to represent a number
 - double-precision – C language **double**
 - Uses 64 bits (1 + 11 + 52) to represent a number
 - actual standard (IEEE 754-2008) adds half-precision float (16 bits) mainly for graphics and neural networks, quadruple-precision (128 bits) and octuple-precision (256 bits) for special scientific computations

The Representation/Encoding of Floating Point Number

- Mantissa encoded as the sign and absolute value (magnitude) – equivalent to the **direct representation**
- Exponent encoded in **biased representation** ($K=+127$ for single precision, $+1023$ for double)
- The implicit leading one can be omitted due to normalization of $m \in \langle 1, 2 \rangle$ – $23+1$ implicit bit for single

$$X = -1^s 2^{A(E)-127} m \quad \text{where } m \in \langle 1, 2 \rangle$$

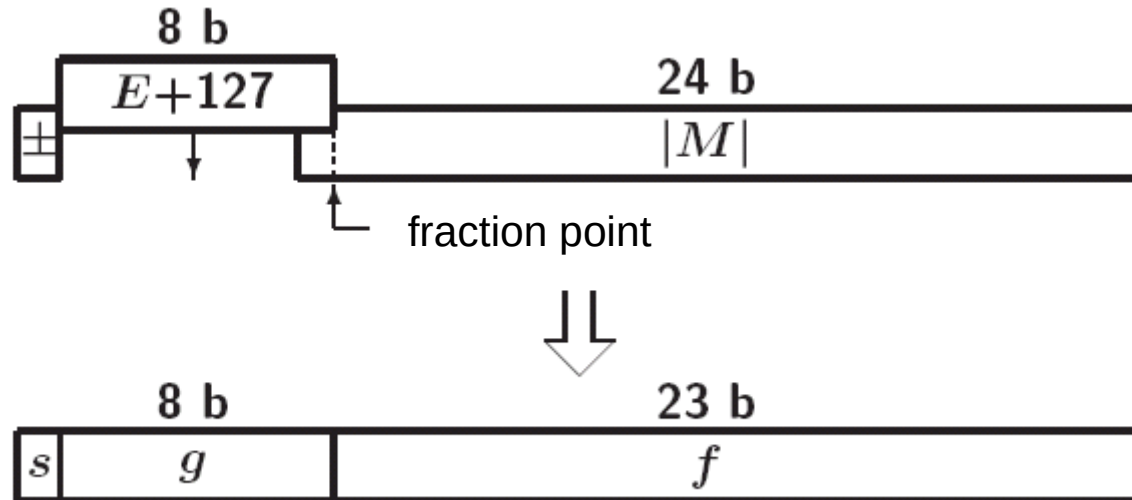
$$m = 1 + 2^{-23} M$$



Radix point position for E and M

ANSI/IEEE Std 754-1985 – 32b and 64b Formats

ANSI/IEEE Std 754-1985 — single precision format — 32b



ANSI/IEEE Std 754-1985 — double precision format — 64b

$g \dots 11b$

$f \dots 52b$

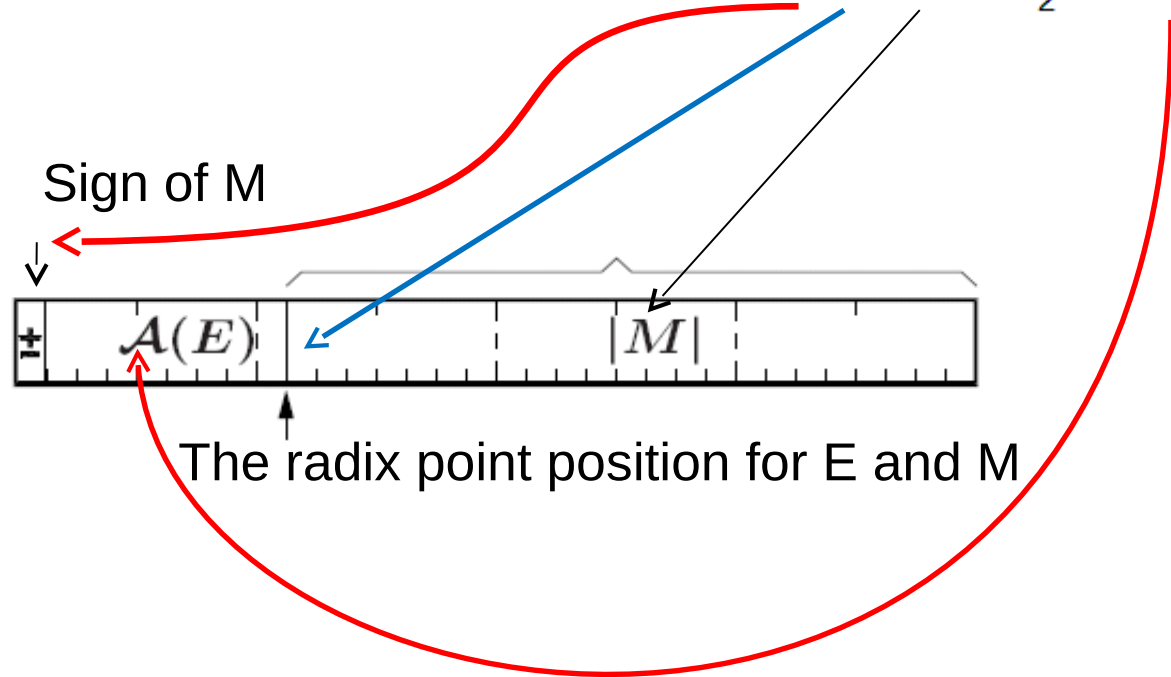
ANSI/IEEE Std 754-1985 — half precision format — 16b

$g \dots 5b$

$f \dots 10b$

Examples of (De)Normalized Numbers in Base 10 and 2

- -2.34×10^{56} ← normalized
 - $+0.002 \times 10^{-4}$ ← not normalized
 - $+987.02 \times 10^9$ ← not normalized
- binary
- $\pm 1.xxxxxxx_2 \times 2^{yyyy}$



IEEE 754 – Conversion Examples

Find IEEE-754 float representation of -12.625_{10}

- Step #1: convert $-12.625_{10} = -1100.101_2$
- Step #2: normalize $-1100.101_2 = -1.100101_2 * 2^3$
- Step #3:

Fill sign field, negative for this case $\rightarrow S=1$.

Exponent + 127 = $3+127 = 130 \rightarrow 1000\ 0010_2$.

The first mantissa bit 1 is a hidden one \rightarrow

1 1000 0010 . 1001 0100 0000 0000 0000 000

Alternative approach, separate sign, find floor of binary logarithm for absolute value, compute equivalent power of two, divide number (result is normalized) and, subtract one, multiply by two, if > 1 subtract and append 1 to result else append 0, multiply by two and repeat.

Example 0.75

$$0.75_{10} = 0.11_2 = 1.1 \times 2^{-1} = 3/4$$

$$1.1 = 1.F \rightarrow F = 1$$

$$E - 127 = -1 \rightarrow E = 127 - 1 = 126 = 01111110_2$$

$$S = 0$$

$$00111111010000000000000000000000 = \\ 0x3F400000$$

Often Inexact Floating Point Number Representation

Decadic number with finite expansion \rightarrow infinite binary expansion

Examples:

$$0.1_{\text{ten}} \rightarrow 0.2 \rightarrow 0.4 \rightarrow 0.8 \rightarrow 1.6 \rightarrow 3.2 \rightarrow 6.4 \rightarrow 12.8 \rightarrow \dots$$

$$\begin{aligned} 0.1_{10} &= 0.00011001100110011\dots_2 \\ &= 0.\underline{00011}_2 \text{ (infinite bit stream)} \end{aligned}$$

More bits only enhance precision of 0.1_{10} representation

Limitation

Only numbers corresponding to $x/2^k$ allows exact representation, all other are **stored inexact**

Value representation

$1/3$ $0.0101010101[01]..._2$

$1/5$ $0.001100110011[0011]..._2$

$1/10$ $0.0001100110011[0011]..._2$

Special Values – Not a Number (NaN) and Infinity

- If the result of the mathematical operation is not defined, such as the calculation of $\log(-1)$, or the result is ambiguous $0/0$, $+\text{Inf} + -\text{Inf}$, then the value NaN (Not-a-Number) is saved
= exponent is set to all ones and the mantissa is nonzero.

NaN

positive	0	11111111	<i>mantissa != 0</i>	NaN
----------	---	----------	----------------------	-----

- If the operation results only overflow the range or infinity is on input ($X + +\text{Inf}$) and result sign is unambiguous

Infinity

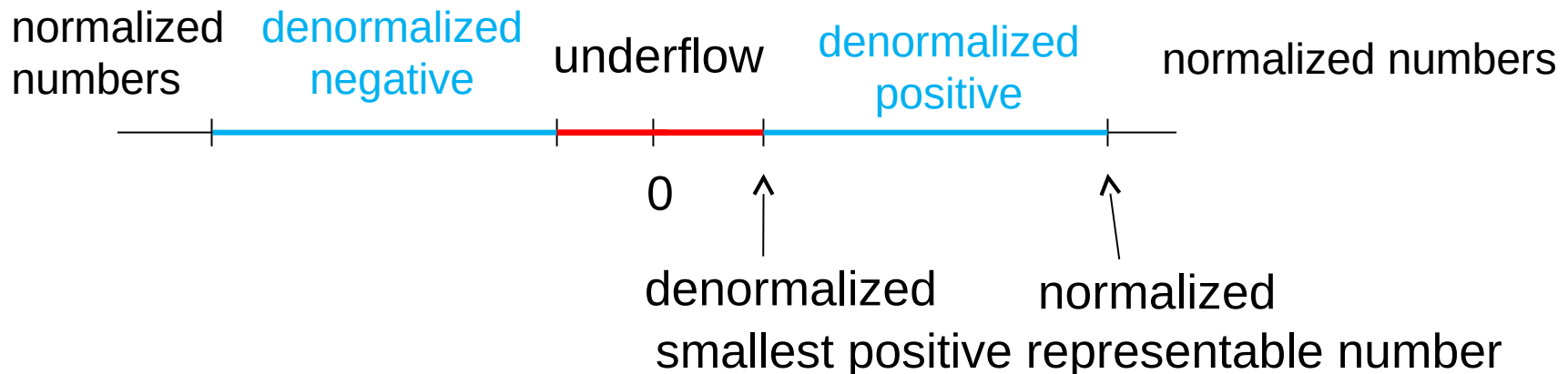
positive	0	11111111	00000000000000000000000000000000	+Inf
negative	1	11111111	00000000000000000000000000000000	-Inf

Implied (Hidden) Leading 1 bit

- Most significant bit of the mantissa is one for each normalized number and it is not stored in the representation for the normalized numbers
- If exponent representation is zero then encoded value is zero or denormalized number which requires to store most significant bit and there is zero considered on usual hidden one location
- Denormalized numbers allow to keep resolution in the range from the smallest normalized number to zero but the computation when some of operands is denormalized is more complex. Some coprocessors do not support denormalized numbers and emulation is required to fulfill IEEE-754 strict requirements, Intel coprocessors supports denormalized numbers

Underflow/Lost of the Precision for IEEE-754 Representation

- The case where stored number value is not zero but it is smaller than smallest number which can be represented in the normalized form
- The direct underflow to the zero can be prevented by extension of the representation range by denormalized numbers



Representation of the Fundamental Values

Zero

Positive zero	0 00000000 000000000000000000000000	+0.0
Negative zero	1 00000000 000000000000000000000000	-0.0

Infinity

Positive infinity	0 11111111 000000000000000000000000	+Inf
Negative infinity	1 11111111 000000000000000000000000	-Inf

Representation corner values

Smallest normalized	* 00000001 000000000000000000000000	$\pm 2^{(1-127)}$ $\pm 1.1755 \cdot 10^{-38}$
Biggest denormalized	* 00000000 111111111111111111111111	$\pm (1 - 2^{-23}) 2^{(1-126)}$
Smallest denormalized	* 00000000 000000000000000000000001	$\pm 2^{-23} 2^{-126}$ $\pm 1.4013 \cdot 10^{-45}$
Max. value	0 11111110 111111111111111111111111	$(2 - 2^{-23}) 2^{(127)}$ $+ 3.4028 \cdot 10^{+38}$

The Table in Another Format

Type	31	28	24	20	16	12	8	4	0	Watch in Windows®	Value
	sign exponent(8)							fraction (23-bit)			
Zero	0	0	0	0	0	0	0	0	0	0.00000000	0
One	0	0	1	1	1	1	1	1	1	1.00000000	1
Minus One	1	0	1	1	1	1	1	1	1	-1.00000000	-1.0
Smallest denormalized number	*	0	0	0	0	0	0	0	0	1.401e-045#DEN	$\pm 2^{-23} \times 2^{-126} = \pm 2^{-149} \approx \pm 1.4 \times 10^{-45}$
"Middle" denormalized number	*	0	0	0	0	0	0	0	0	5.877e-039#DEN	$\pm 2^{-1} \times 2^{-126} = \pm 2^{-127} \approx \pm 5.88 \times 10^{-39}$
Largest denormalized number	*	0	0	0	0	0	0	0	0	1.175e-038#DEN	$\pm (1 - 2^{-23}) \times 2^{-126} \approx \pm 1.18 \times 10^{-38}$
Smallest normalized number	*	0	0	0	0	0	0	0	0	1.1754944e-038	$\pm 2^{-126} \approx \pm 1.18 \times 10^{-38}$
Largest normalized number	*	1	1	1	1	1	1	1	1	3.4028235e+038	$\pm (2 - 2^{-23}) \times 2^{127} \approx \pm 3.4 \times 10^{38}$
Positive infinity	0	1	1	1	1	1	1	1	1	1.#INF000	$+\infty$
Negative infinity	1	1	1	1	1	1	1	1	1	-1.#INF000	$-\infty$
Not a number	*	1	1	1	1	1	1	1	1	1.#QNaN00	NaN

* Sign bit can be either 0 or 1.

Figure: Floating-point Binary

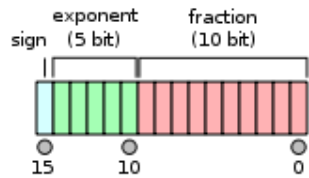
Copyright libg.org

Some Features of ANSI/IEEE Standard Floating-point Formats

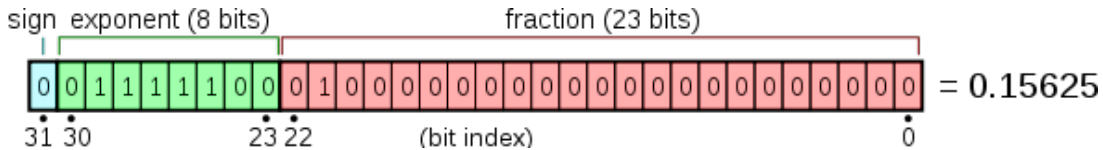
Feature	Single/Float	Double/Long
Word width in bits	32	64
Significand in bits	23 + 1 hidden	52 + 1 hidden
Significand range	$[1, 2 - 2^{-23}]$	$[1, 2 - 2^{-52}]$
Exponent bits	8	11
Exponent bias	127	1023
Zero (± 0)	$e + \text{bias} = 0, f = 0$	$e + \text{bias} = 0, f = 0$
Denormal	$e + \text{bias} = 0, f \neq 0$ represents $\pm 0.f \cong 2^{-126}$	$e + \text{bias} = 0, f \neq 0$ represents $\pm 0.f \cong 2^{-1022}$
Infinity ($\cong \infty$)	$e + \text{bias} = 255, f = 0$	$e + \text{bias} = 2047, f = 0$
Not-a-number (NaN)	$e + \text{bias} = 255, f \neq 0$	$e + \text{bias} = 2047, f \neq 0$
Ordinary number	$e + \text{bias} \in [1, 254]$ $e \in [-126, 127]$ represents $1.f \cong 2^e$	$e + \text{bias} \in [1, 2046]$ $e \in [-1022, 1023]$ represents $1.f \cong 2^e$
<i>min</i>	$2^{-126} \circ 1.2 \cong 10^{-38}$	$2^{-1022} \circ 2.2 \cong 10^{-308}$
<i>max</i>	$\circ 2^{128} \circ 3.4 \cong 10^{38}$	$\circ 2^{1024} \circ 1.8 \cong 10^{308}$

IEEE-754 Formats

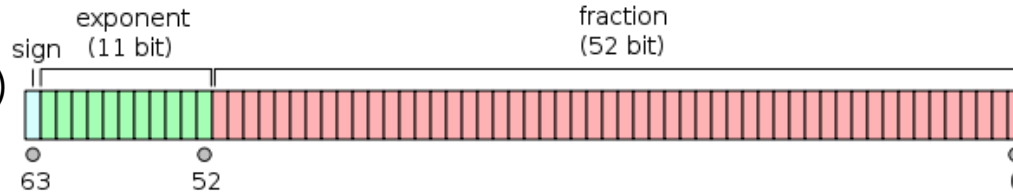
Half precision (binary16)



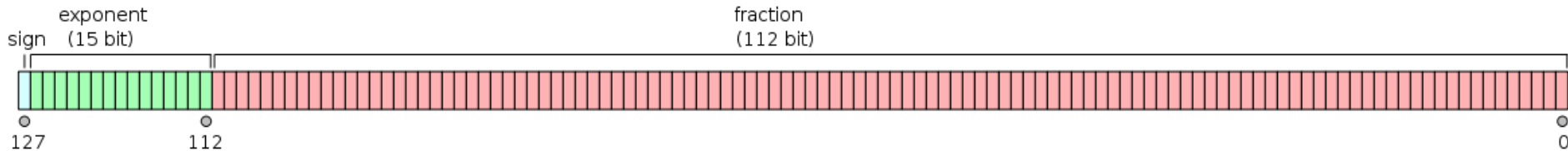
Single precision (binary32)



Double precision (binary64)

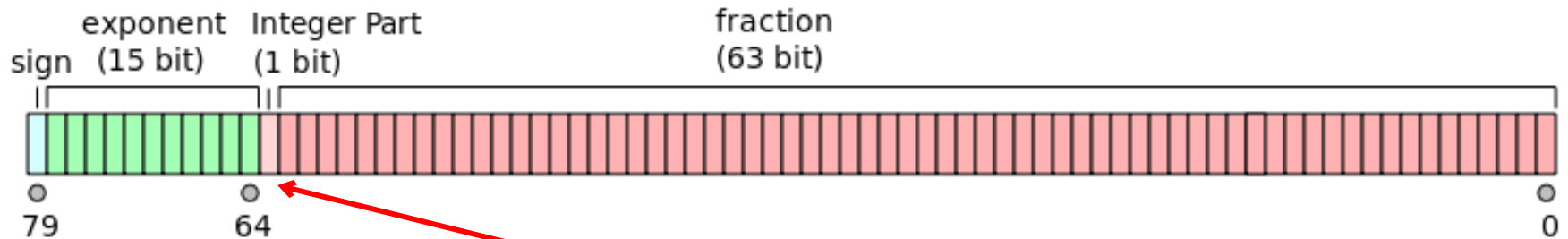


Quadruple precision (binary128)



Source: Herbert G. Mayer, PSU

X86 Extended Precision Format (80-bits)



Bit 1. is not hidden in mantissa!

Advanced readers note:

- *Intel processors integrate arithmetic coprocessor on the single chip with processor (from Intel 80486), which computes float and double expressions in „extended precision“ internally and the results are rounded to float/double when stored.*
- *But Streaming SIMD Extensions (SSE) instructions (vector operations) from Intel Pentium III on provides only double precision and the result rounding/precision can be dependent on compiler selection*

IEEE-754 Special Values Summary

sign bit	Exponent representation	Mantissa	Represented value/meaning
0	$0 < e < 255$	any value	normalized positive number
1	$0 < e < 255$	any value	normalized negative number
0	0	> 0	denormalized positive number
1	0	> 0	denormalized negative number
0	0	0	positive zero
1	0	0	negative zero
0	255	0	positive infinity
1	255	0	negative infinity
0	255	$\neq 0$	NaN – does not represent a number
1	255	$\neq 0$	NaN – does not represent a number

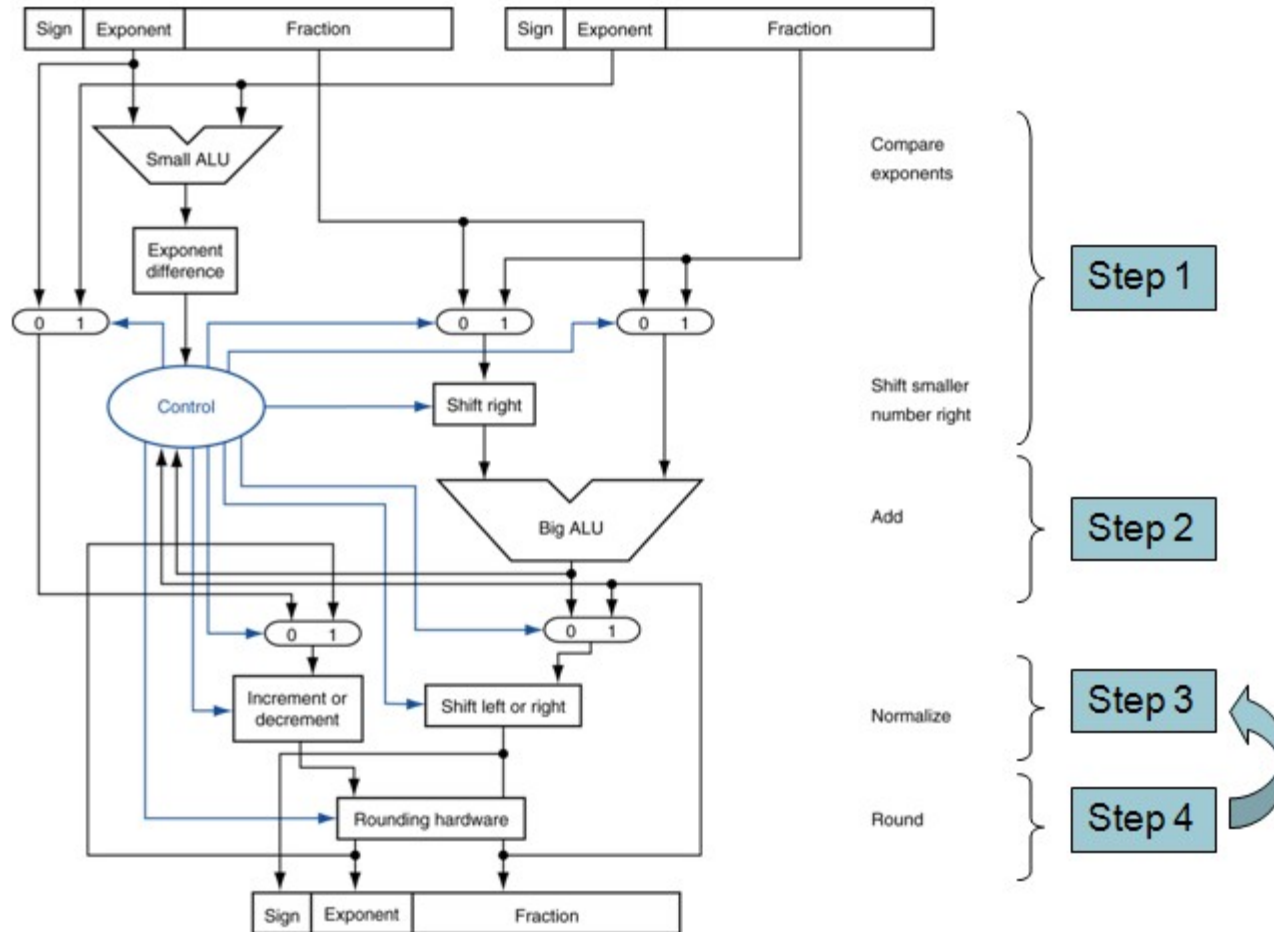
Comparison

- Comparison of the two IEEE-754 encoded numbers requires to solve signs separately but then it can be processed by unsigned ALU unit on the representations
$$A \geq B \iff A - B \geq 0 \iff D(A) - D(B) \geq 0$$
- This is advantage of the selected encoding and reason why sign is not placed at start of the mantissa

Addition of Floating Point Numbers

- The number with bigger exponent value is selected
- Mantissa of the number with smaller exponent is shifted right – the mantissas are then expressed at same scale
- The signs are analyzed and mantissas are added (same sign) or subtracted (smaller number from bigger)
- The resulting mantissa is shifted right (max by one) if addition overflows or shifted left after subtraction until all leading zeros are eliminated
- The resulting exponent is adjusted according to the shift
- Result is normalized after these steps
- The special cases and processing is required if inputs are not regular normalized numbers or result does not fit into normalized representation

Hardware of the Floating Point Adder



Multiplication of Floating Point Numbers

- Exponents are added and signs xor-ed
- Mantissas are multiplied
- Result can require normalization
 - max 2 bits right for normalized numbers
- The result is rounded

- Hardware for multiplier is of the same or even lower complexity as the adder hardware – only adder part is replaced by unsigned multiplier

Floating Point Arithmetic Operations Overview

Addition: $A \cdot z^a, B \cdot z^b, b < a$ unify exponents
 $B \cdot z^b = (B \cdot z^{b-a}) \cdot z^{b-(b-a)}$ by shift of mantissa
 $A \cdot z^a + B \cdot z^b = [A + (B \cdot z^{b-a})] \cdot z^a$ sum + normalization

Subtraction: unification of exponents, subtraction and normalization

Multiplication: $A \cdot z^a \cdot B \cdot z^b = A \cdot B \cdot z^{a+b}$
 $A \cdot B$ - normalize if required
 $A \cdot B \cdot z^{a+b} = A \cdot B \cdot z \cdot z^{a+b-1}$ - by left shift

Division: $A \cdot z^a / B \cdot z^b = A/B \cdot z^{a-b}$
 A/B - normalize if required
 $A/B \cdot z^{a-b} = A/B \cdot z \cdot z^{a-b+1}$ - by right shift

*Multi-byte Numbers

and their store in computer memory

Motivation: What is the output of next code snippet?

```
#include <stdio.h>
int main() {
    unsigned char p[] = {0, 0, 0xa0, 0xbf};
    printf("%i = %u = %f\n", *(int*)p,
           *(unsigned int *)p, *(float *)p);
}
```

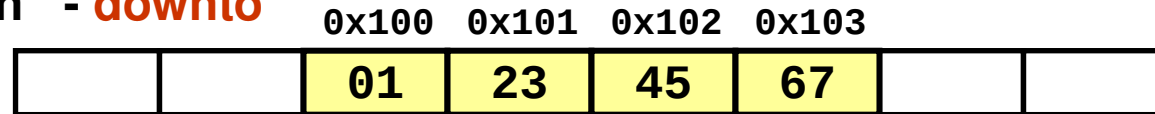
What is output of this code?

- A) it cannot be compiled, syntax error
- B) -65 = 160 = 0.0
- C) -1080033280 = 3214934016 = -1.250000

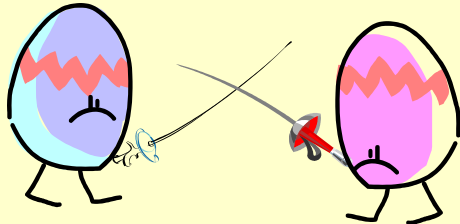
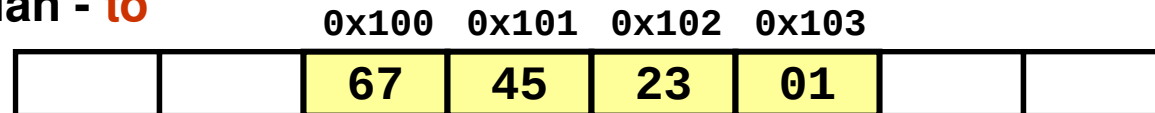
How to Store Multi-byte Number in Memory

Hexadecimal number: 0x1234567

Big Endian - downto



Little Endian - to



Little-Endien comes from a book by Gulliver's Travels, Jonathan Swift 1726, in which he referred to one of the two opposing factions of the Lilliput. Ones ate eggs from the narrow end to the broader while Big Endien proceeded the other way around. And the war did not wait long ...

Do you remember how war ended?



Do you understand little endien?

```
#include <stdio.h>
int main() {
    unsigned char p[] = {0, 0, 0xa0, 0xbf};
    *(int*)p=10;
    printf("%02x,%02x,%02x,%02x\n", p[0],p[1],p[2],p[3]);
}
```

What is output of this code?

- A) it cannot be compiled, syntax error
- B) 0a,00,00,00
- C) 00,00,00,0a
- D) 00,00,0a,00
- E) 00,0a,00,00

Memory Alignment (cz:zarovnání paměti?)

.align n directive

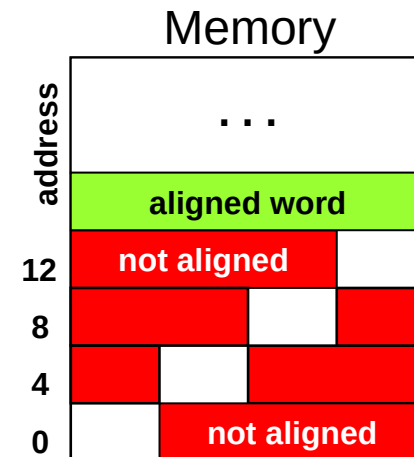
- next space allocated for data or text starts at 2^n divisible address

Example **.align 2**

- two least significant bits (LSB) are equal to **00**

Memory is addressed as **byte array** as usually (in C more precisely as array of **chars**)

The word of 32-bit processor is formed of 4-bytes in such case



Align in Data Segment Filled by Assembler

```
.data
.align 2 // or .align 4 on x86, use .p2align and .baling
var1: .byte 3, 5, 'A', 'P', '0'
.align 2 // or .align 4 on x86, use .p2align and .baling
var2: .word 0x12345678 // or .long on x86
.align 3 // or .align 8 on x86, use .p2align and .baling
var3: .2byte 1000 // or .word on x86
```

var1 ↘

var2 ↘

BIG ENDIAN	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0x2000	3	5	41	50	4F				12	34	56	78				
0x2010	10	00														

var3 ↗
var1 ↘

var2 ↘

LITTLE ENDIAN	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0x2000	3	5	41	50	4F				78	56	34	12				
0x2010	00	10														

var3 ↗

C Language: Pointer

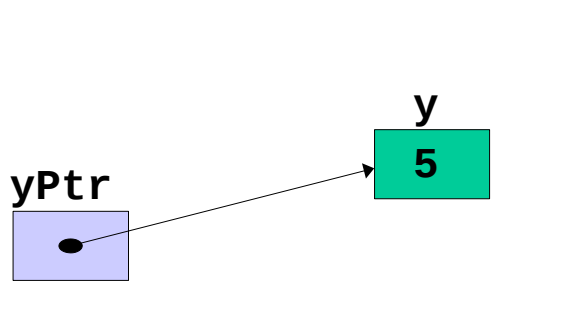
& (address operator)

Returns the lowest address in memory address space where space/cells allocated to store variable starts.

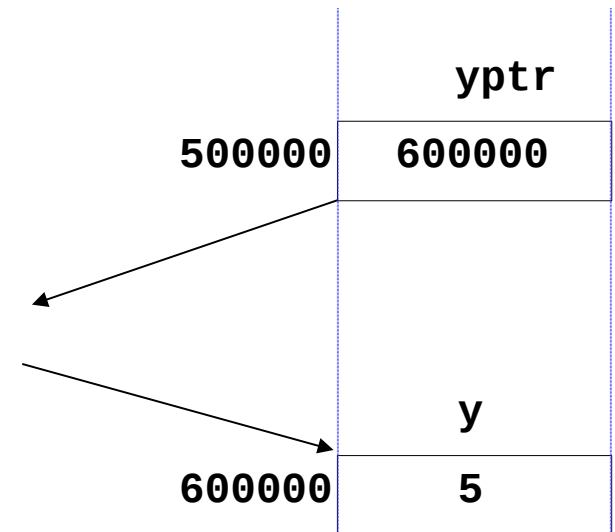
Example

```
int y = 5;  
int *yPtr;  
yPtr = &y;           // yPtr is signed to y address
```

yPtr "points to" y



y
addressbecom
es value of
yptr



C Language: Pointer Operations

& (address operator)

returns address of operand

***** dereference address

returns value stored on address interpreted according to pointer type

***** and **&** are inverse
(but are not applicable in each case)

$*\&\text{myVar} == \text{myVar}$

and

$\&*yPtr == yPtr$

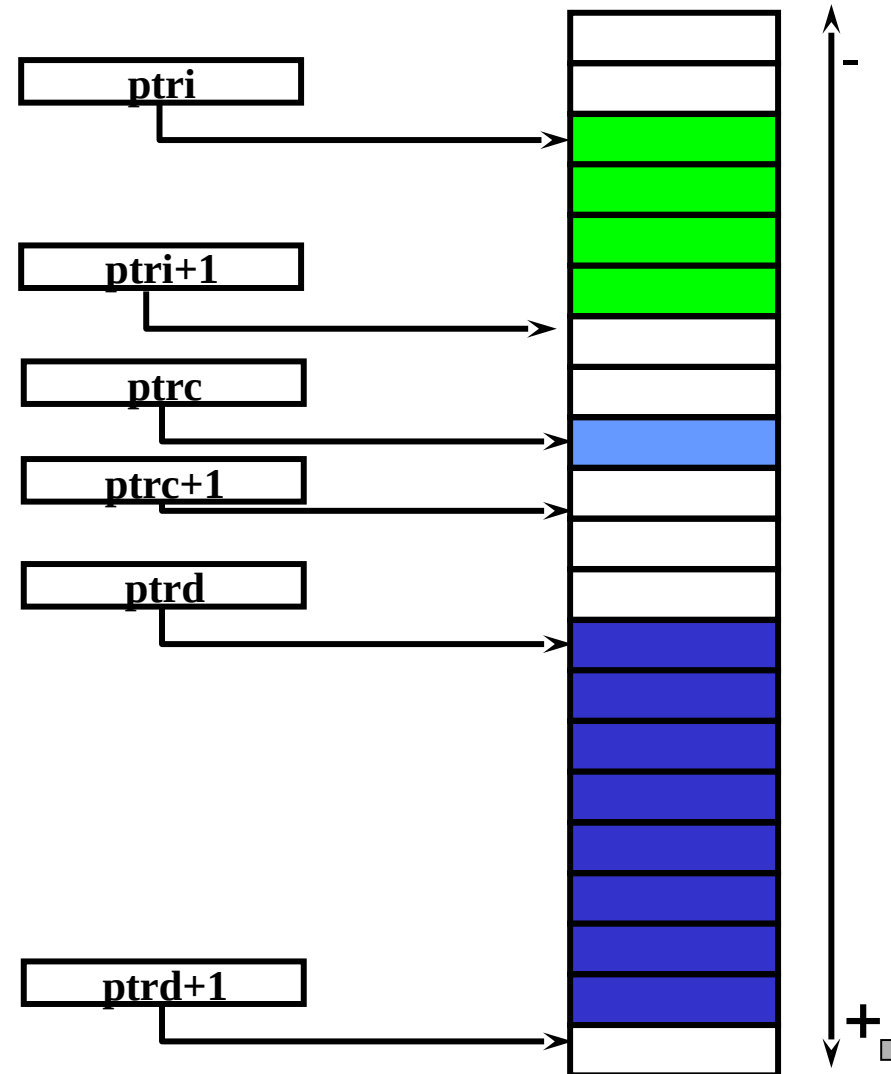
C Language: Size of Element Pointed by C Pointer

```
int * ptri;  
char * ptrc;  
double * ptrd;
```

```
*ptrx ≡ ptrx[0]  
*(ptrx+1) ≡ ptrx[1]  
*(ptrx+n) ≡ ptrx[n]  
*(ptrx-n) ≡ ptrx[-n]
```

```
nr1 = sizeof (double);  
nr2 = sizeof (double *);  
nr3 = sizeof (int *)
```

nr1 != nr2 but nr2==nr3



C Language: Pointer with **const** Qualifier

```
int x, y;
```

```
int * lpio = &y;
```

```
*lpio = 1; x=*lpio; lpio++;
```

```
const int * lpCio = &y;
```

```
*lpCio = 1; x=*lpCio; lpCio++;
```

```
int * const lpioC = &y;
```

```
*lpioC = 1; x=*lpioC; lpioC++;
```

```
const int * const lpCioC = &y;
```

```
*lpCioC = 1; x=*lpCioC; lpCioC++;
```


C Language and Pointers

```
int i;  
int *p;  
p=&i;
```

```
i=i+1;  
*p=*p+1;  
i++;  
(*p)++;  
p[0]++;
```

```
int pole[30];  
p=pole;  
p=&pole[0];
```

```
for(i=0;i<30;i++)  
pole[i]++;
```

```
p=pole;  
for(i=0;i<30;i++){  
  (*(p++))++;  
}
```

```
p++;  
p=(int*) ((char*)p + sizeof(int));
```

The Lecture and Real Programming Question

Quick Quiz 1.: Is the result of both code fragments a same?

Quick Quiz 2.: Which of the code fragments is processed faster and why?

A:

```
int matrix[M][N];  
int i, j, sum = 0;  
...  
for(i=0; i<M; i++)  
    for(j=0; j<N; j++)  
        sum += matrix[i][j];
```

B:

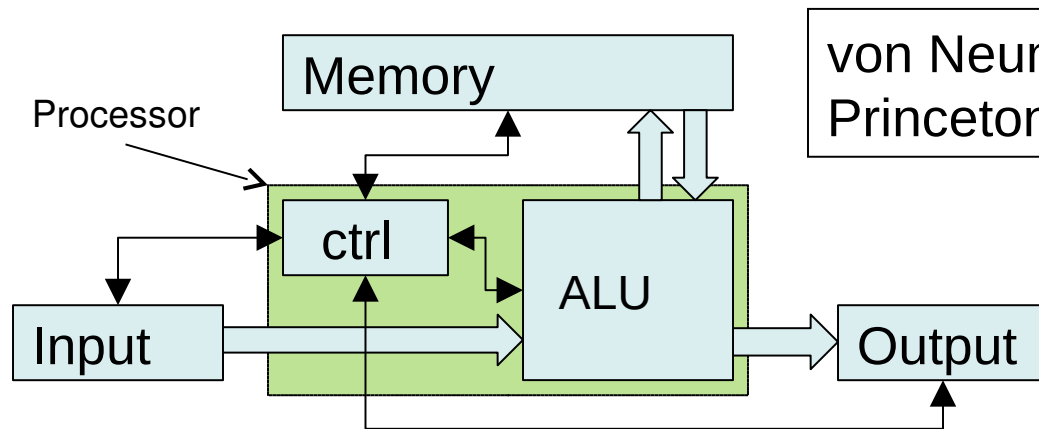
```
int matrix[M][N];  
int i, j, sum = 0;  
...  
for(j=0; j<N; j++)  
    for(i=0; i<M; i++)  
        sum += matrix[i][j];
```

Is there a rule how to iterate over matrix element efficiently?

*Memory and Data

and their store in computer memory

John von Neumann Computer Block Diagram



von Neumann's computer architecture
Princeton Institute for Advanced Studies

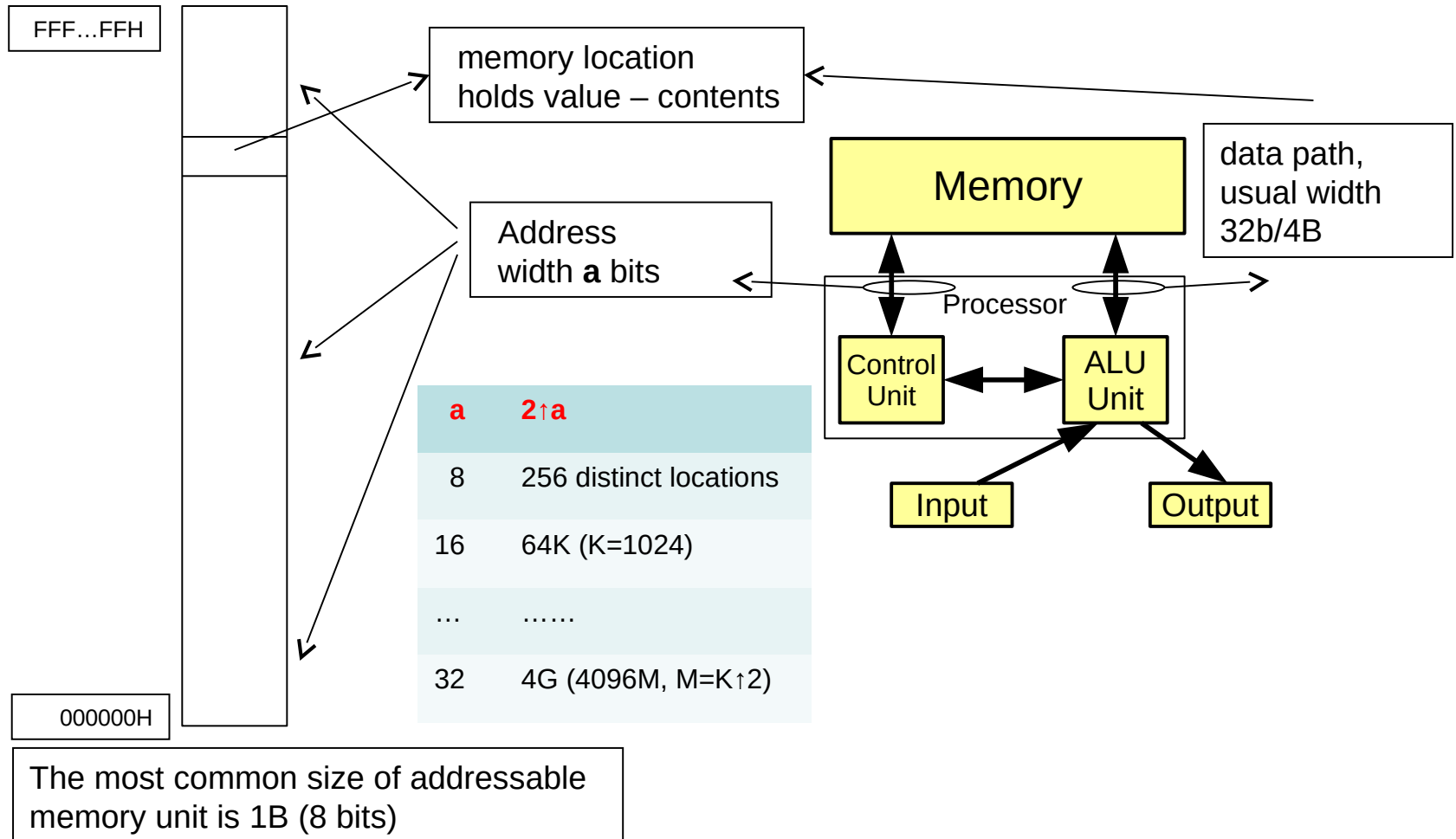


**28. 12. 1903 -
8. 2. 1957**

- 5 functional units – control unit, arithmetic logic unit, memory, input (devices), output (devices)
- An computer architecture should be independent of solved problems. It has to provide mechanism to load program into memory. The program controls what the computer does with data, which problem it solves.
- Programs and results/data are stored in the same memory. That memory consists of a cells of same size and these cells are sequentially numbered (address).
- The instruction which should be executed next, is stored in the cell exactly after the cell where preceding instruction is stored (exceptions branching etc.).
- The instruction set consists of arithmetics, logic, data movement, jump/branch and special/control instructions.

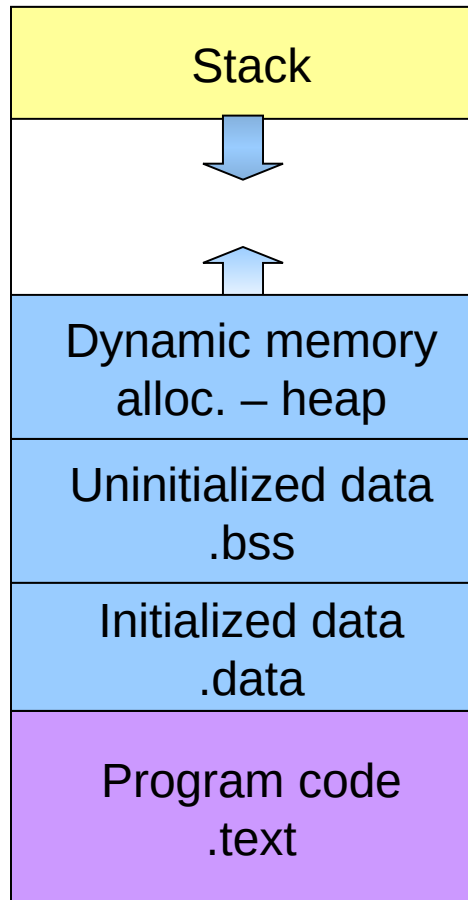
Memory Address Space

It is an array of addressable units (locations) where each unit can hold a data value. Number/range of addresses same as addressable units/words are limited in size.



Program Layout in Memory at Process Startup

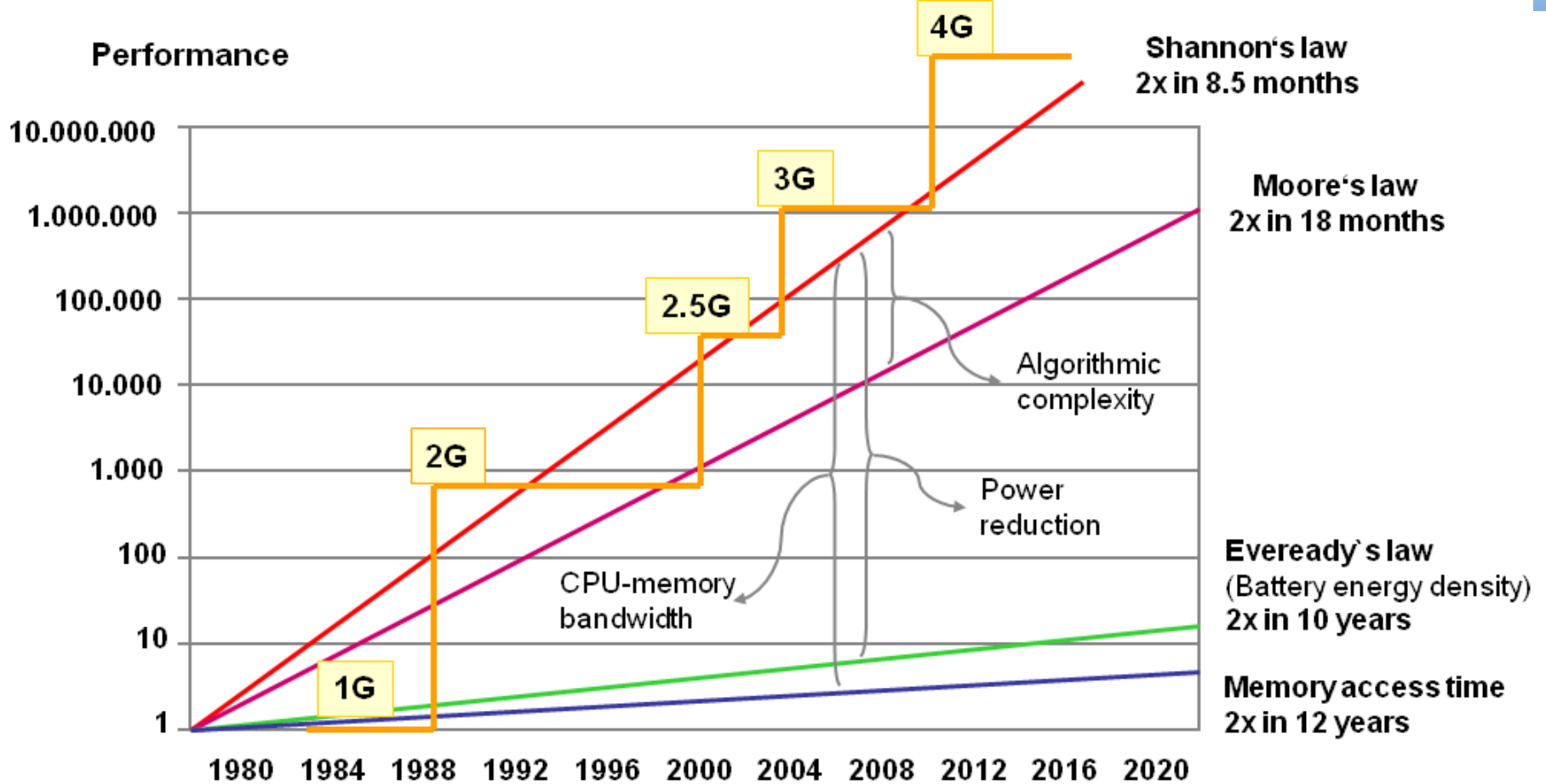
0x7fffffff



0x00000000

- The executable file is mapped (“loaded”) to process address space – sections **.data** and **.text** (note: LMA != VMA for some special cases)
- Uninitialized data area (**.bss** – block starting by symbol) is reserved and zeroed for C programs
- Stack pointer is set and control is passed to the function **_start**
- Dynamic memory is usually allocated above **_end** symbol pointing after **.bss**

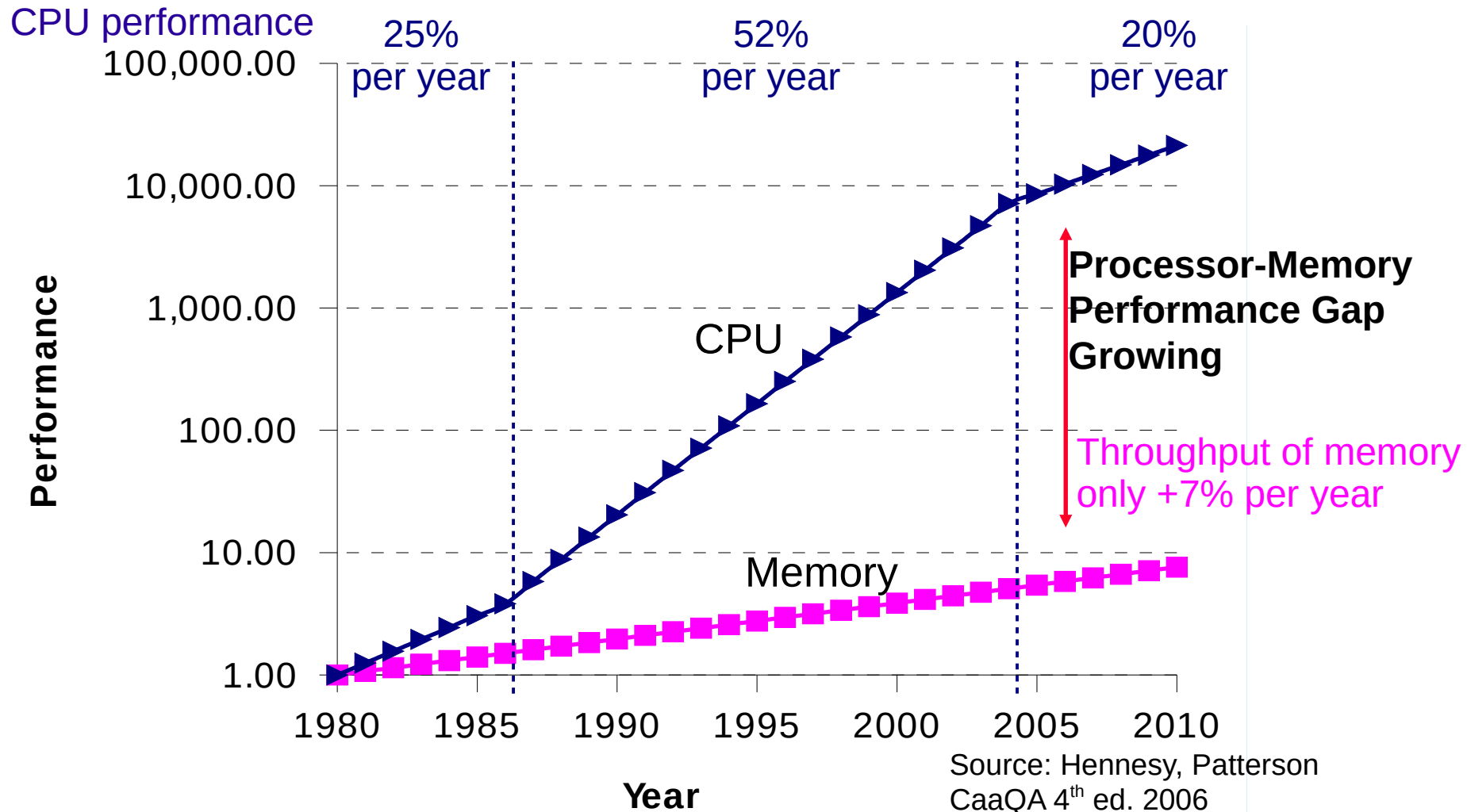
Key Technological Gaps Prediction



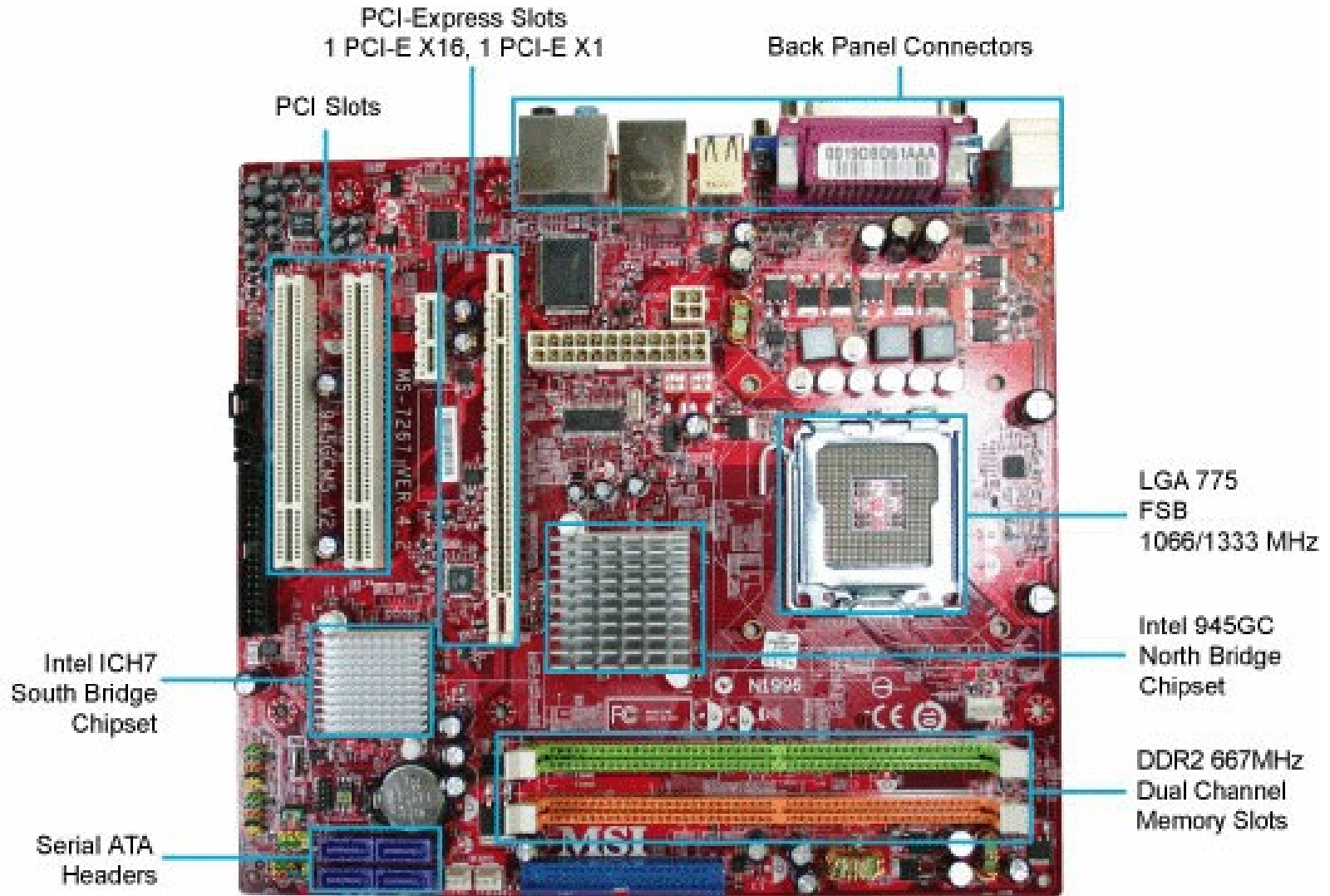
Source: Jan M. Rabaey

Note: The increase in complexity of algorithms over time has been formalized in literature as the so-called Shannon's Law of Algorithmic Complexity.

Memory and CPU Speed – Moore's Law



PC Computer Motherboard

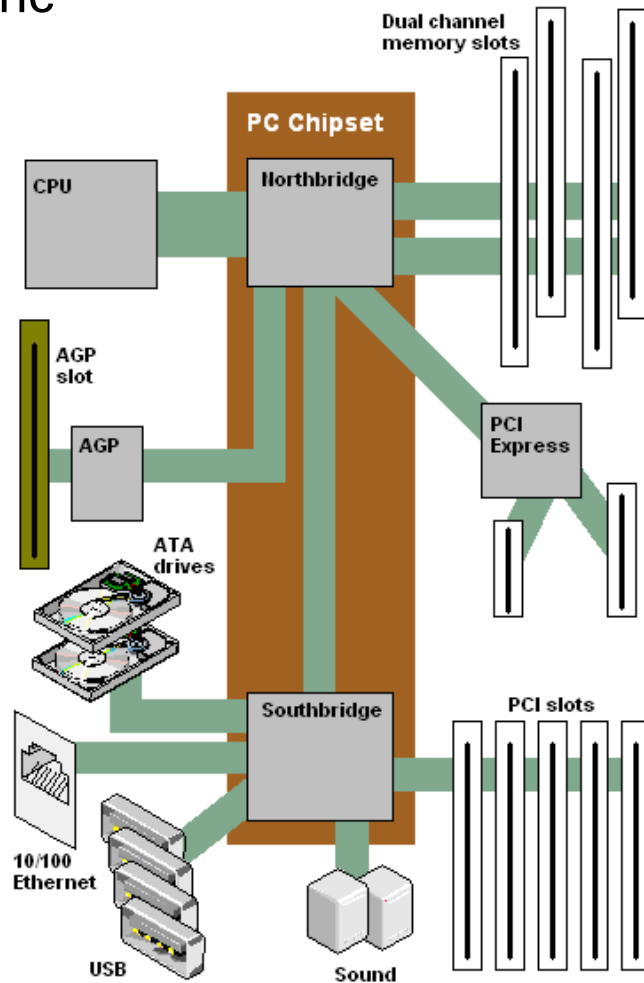


Computer Architecture (Desktop x86 PC)

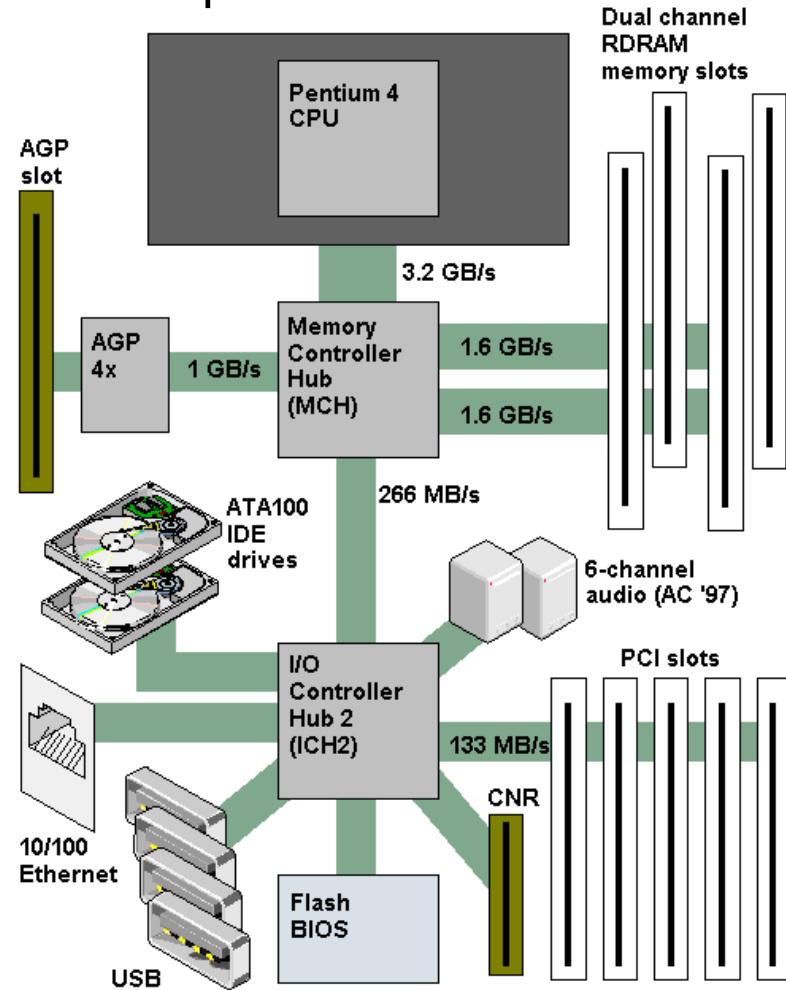
From Computer Desktop Encyclopedia
© 2005 The Computer Language Co., Inc.

From Computer Desktop Encyclopedia
© 2001 The Computer Language Co., Inc.

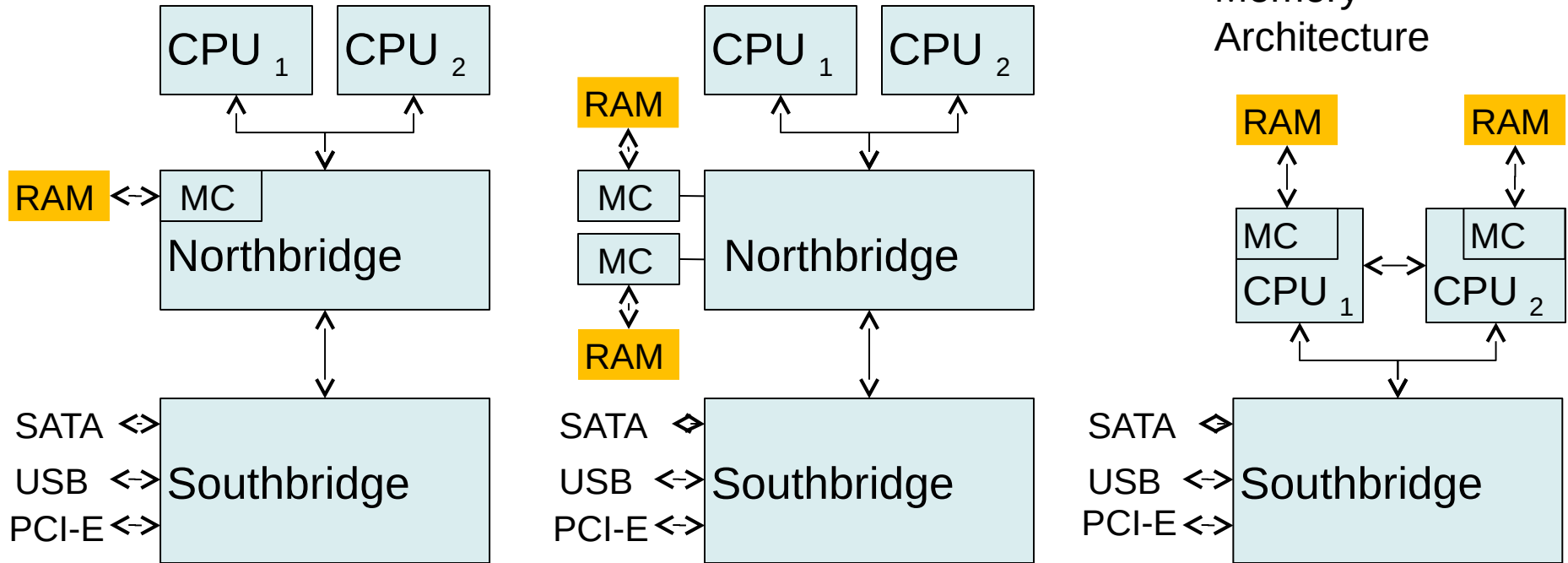
generic



example

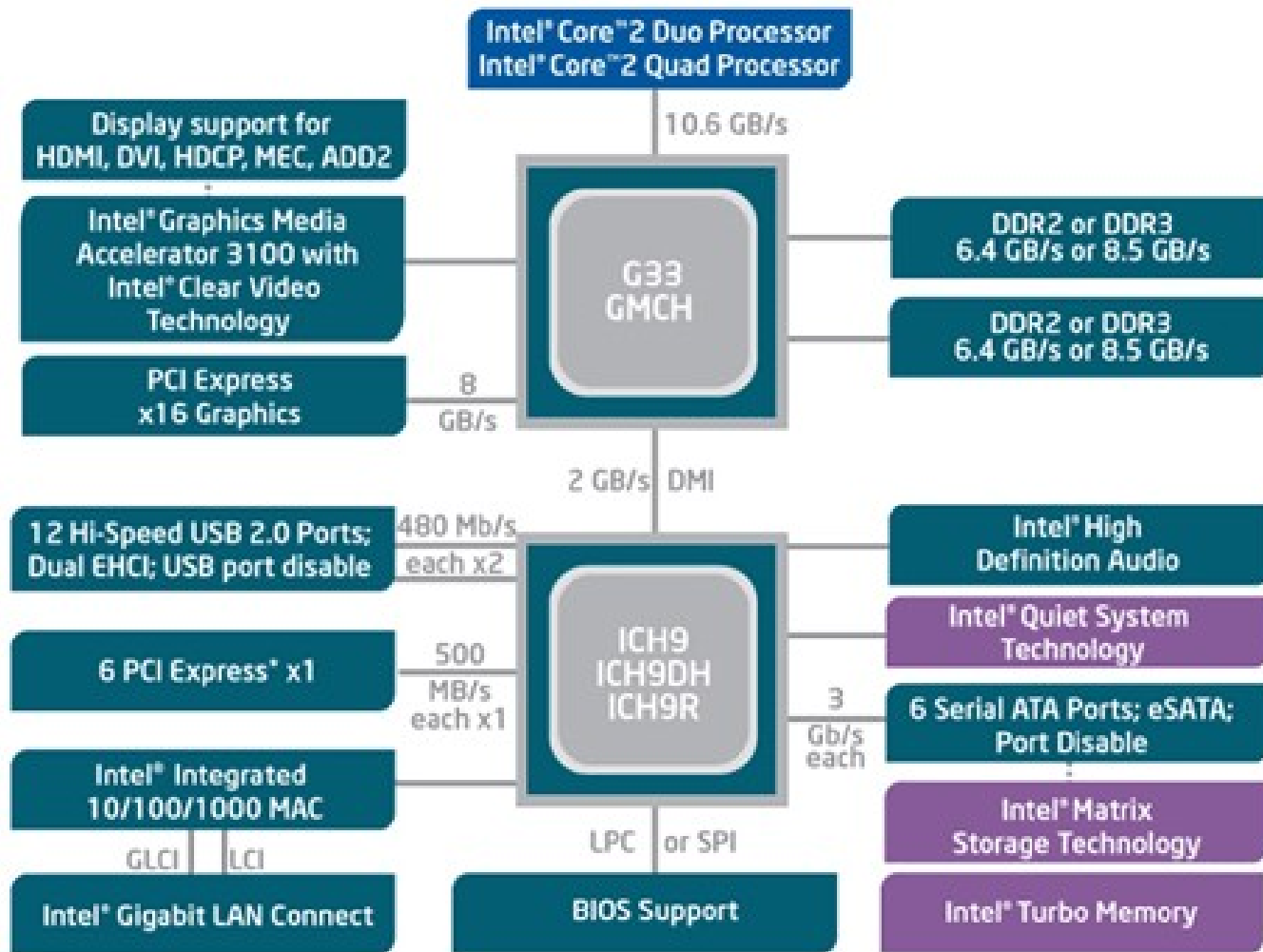


From UMA to NUMA Development (Even in PC Segment)



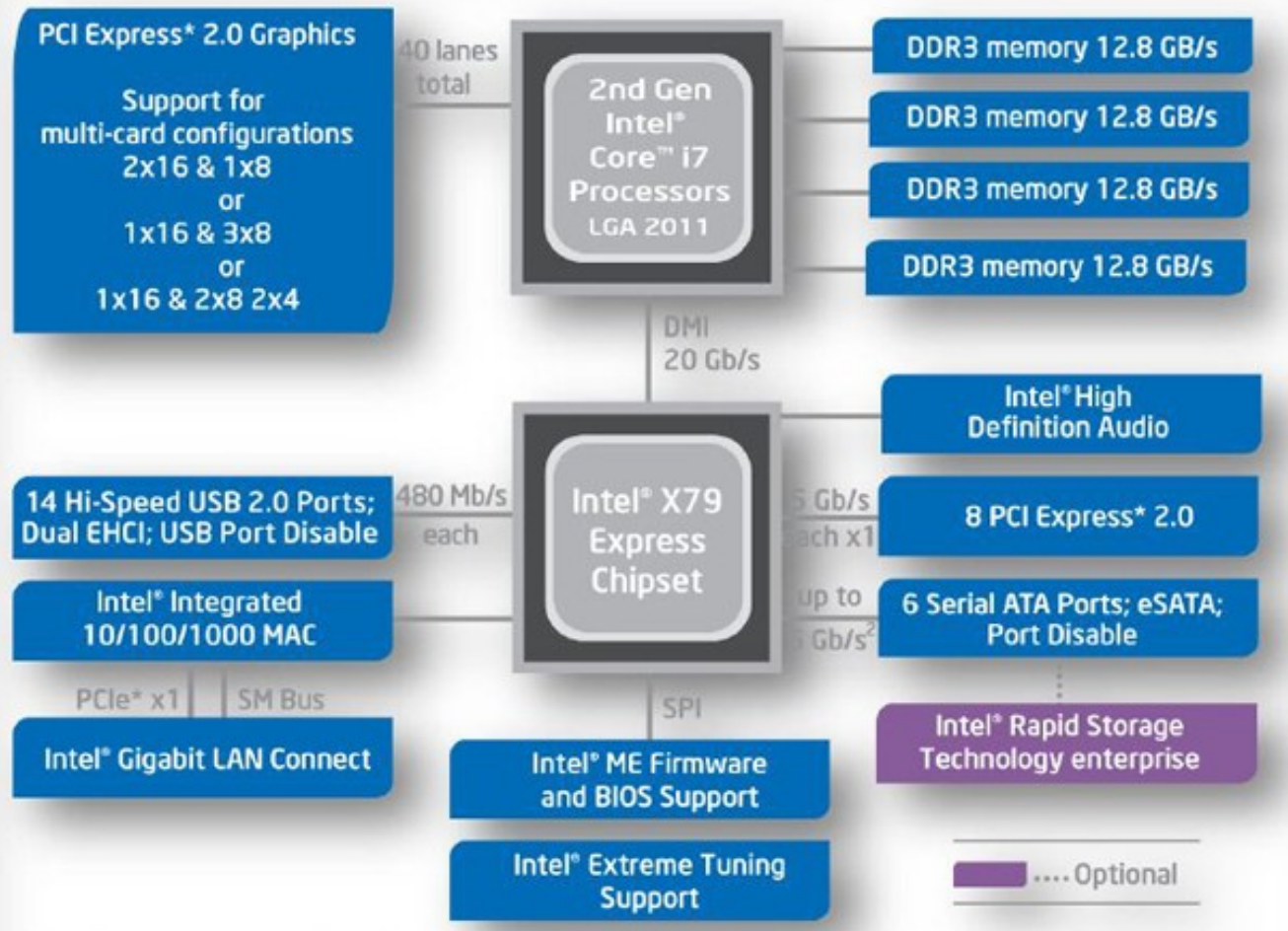
MC - Memory controller – contains circuitry responsible for SDRAM read and writes. It also takes care of refreshing each memory cell every 64 ms.

Intel Core 2 Generation



Northbridge became Graphics and Memory Controller Hub (GMCH)

Intel i3/5/7 Generation



¹Theoretical maximum bandwidth

²All SATA ports capable of 3 Gb/s. 2 ports capable of 6 Gb/s.

Intel® X79 Express Chipset Block Diagram

Memory Subsystem – Terms and Definitions

- Memory address – fixed-length sequences of bits or index
- Data value – the visible content of a memory location

Memory location can hold even more control/bookkeeping information

- validity flag, parity and ECC bits etc.
- Basic memory parameters:
 - Access time – delay or latency between a request and the access being completed or the requested data returned
 - Memory latency – time between request and data being available (does not include time required for refresh and deactivation)
 - Throughput/bandwidth – main performance indicator. Rate of transferred data units per time.
 - Maximal, average and other latency parameters

Memory Types and Maintenance

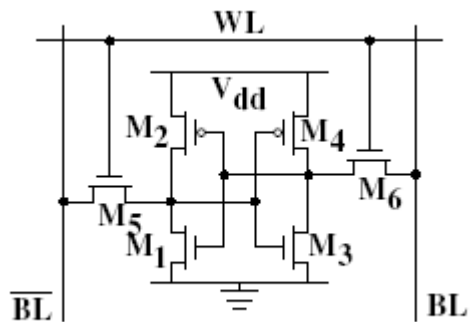
- Types: RWM (RAM), ROM, FLASH
- Implementation: SRAM, DRAM
- Data retention time and conditions (volatile/nonvolatile)
- Dynamic memories (DRAM, SDRAM) require specific care
 - Memory refresh – state of each memory cell has to be internally read, amplified and fed back to the cell once every refresh period (usually about 60 ms), even in idle state. Each refresh cycle processes one row of cells.
 - Precharge – necessary phase of access cycle to restore cell state after its partial discharge by read
 - Both contribute to maximal and average access time.

Typical Memory Parameters

- Memory types: RWM (RAM), ROM, FLASH,
- RAM realization:
SRAM (static), **DRAM** (dynamic).
- RAM = *Random Access Memory*

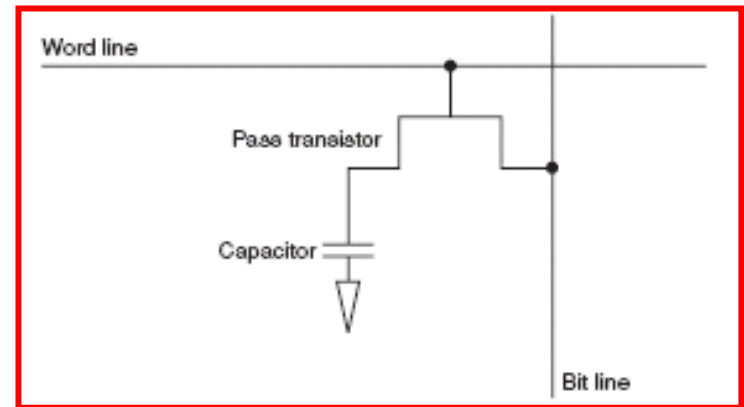
type	transistors per cell	1 bit area	data availability	latency
SRAM	cca 6	$< 0,1 \mu\text{m}^2$	always	$< 1\text{ns} - 5\text{ns}$
DRAM	1	$< 0,001 \mu\text{m}^2$	requires refresh	today $20\text{ ns} - 35\text{ ns}$

Detail of static and Dynamic Memory Bit Cell



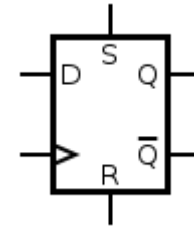
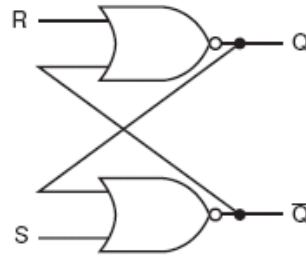
6 transistor static memory cell (single bit)

Single transistor cell of dynamic memory

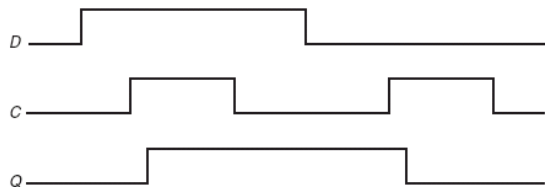
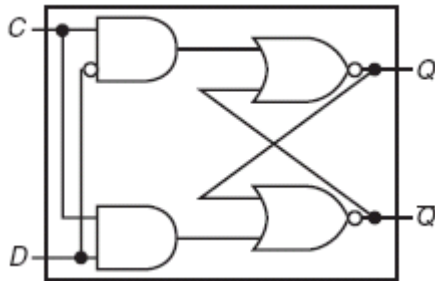


Flip-flop Circuits

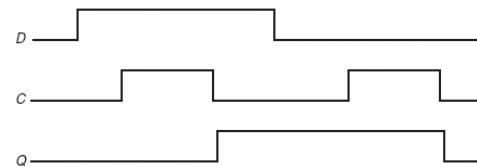
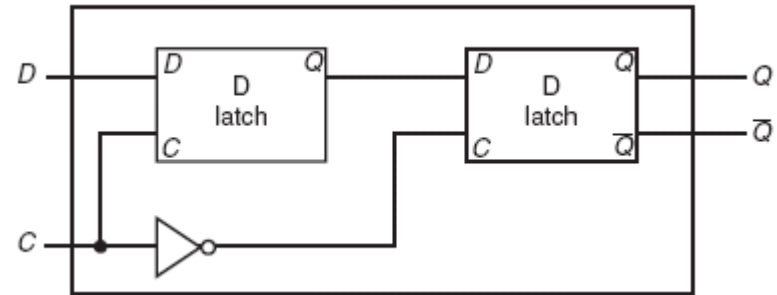
RS



D latch, level-controlled flip-flop

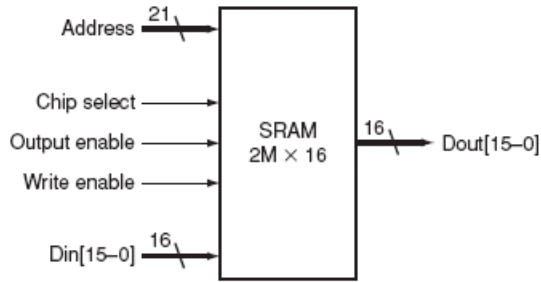


D flip-flop, edge-controlled flip-flop

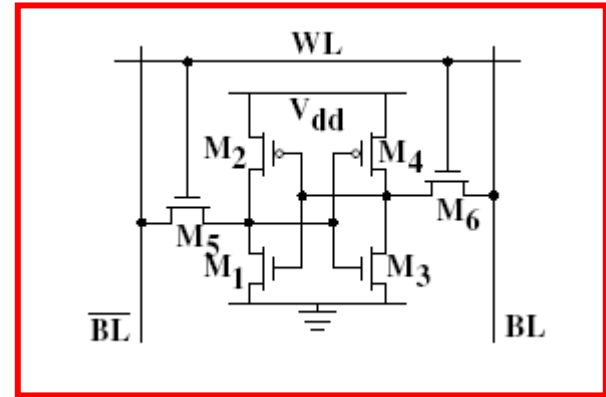


Usual SRAM Chip and SRAM Cell

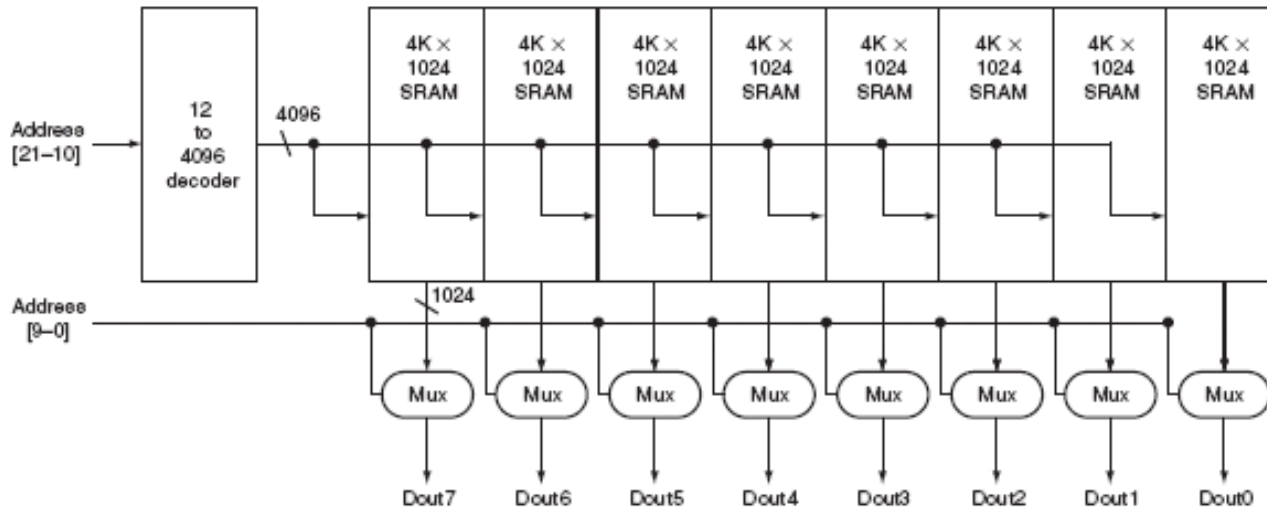
Usual SRAM chip



SRAM memory cell
CMOS technology

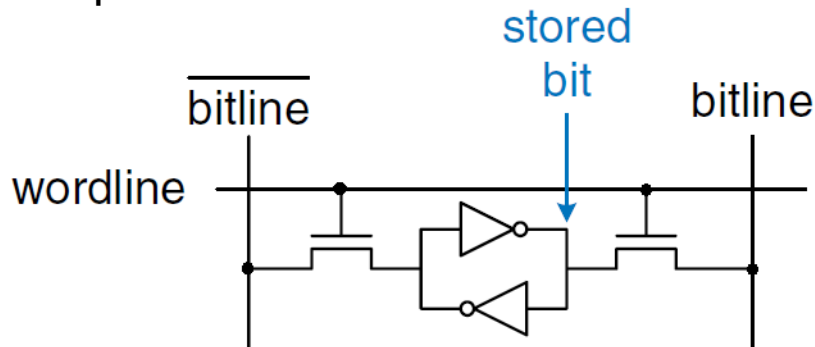


Bigger memory size?



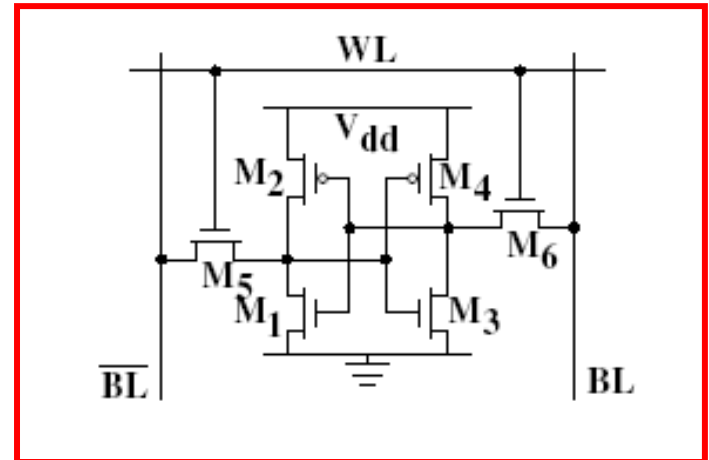
Usual Static Memory Chip Cell

Principle:

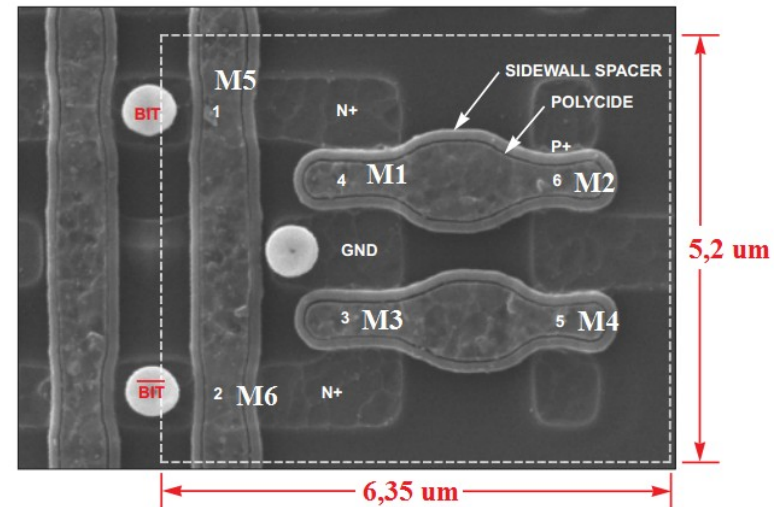
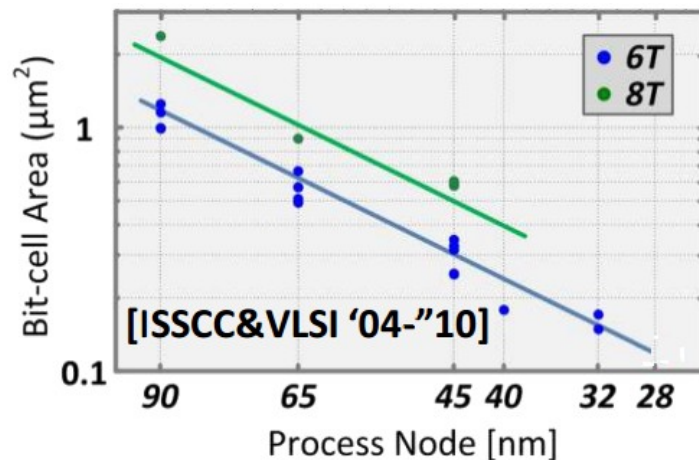


SRAM memory cell

6-transistors CMOS, 4 trans. Version exists

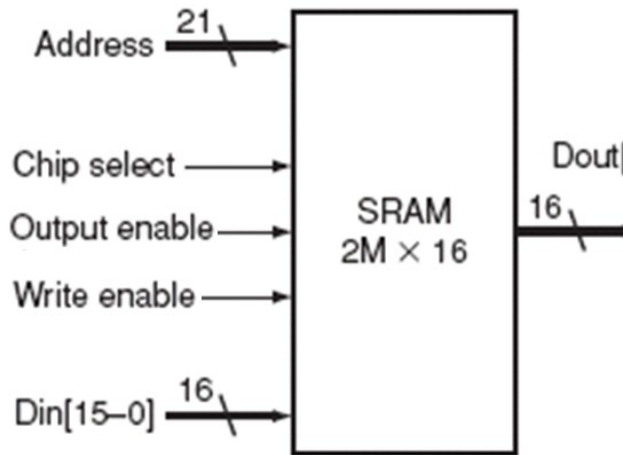


Area of one memory cell(bit):

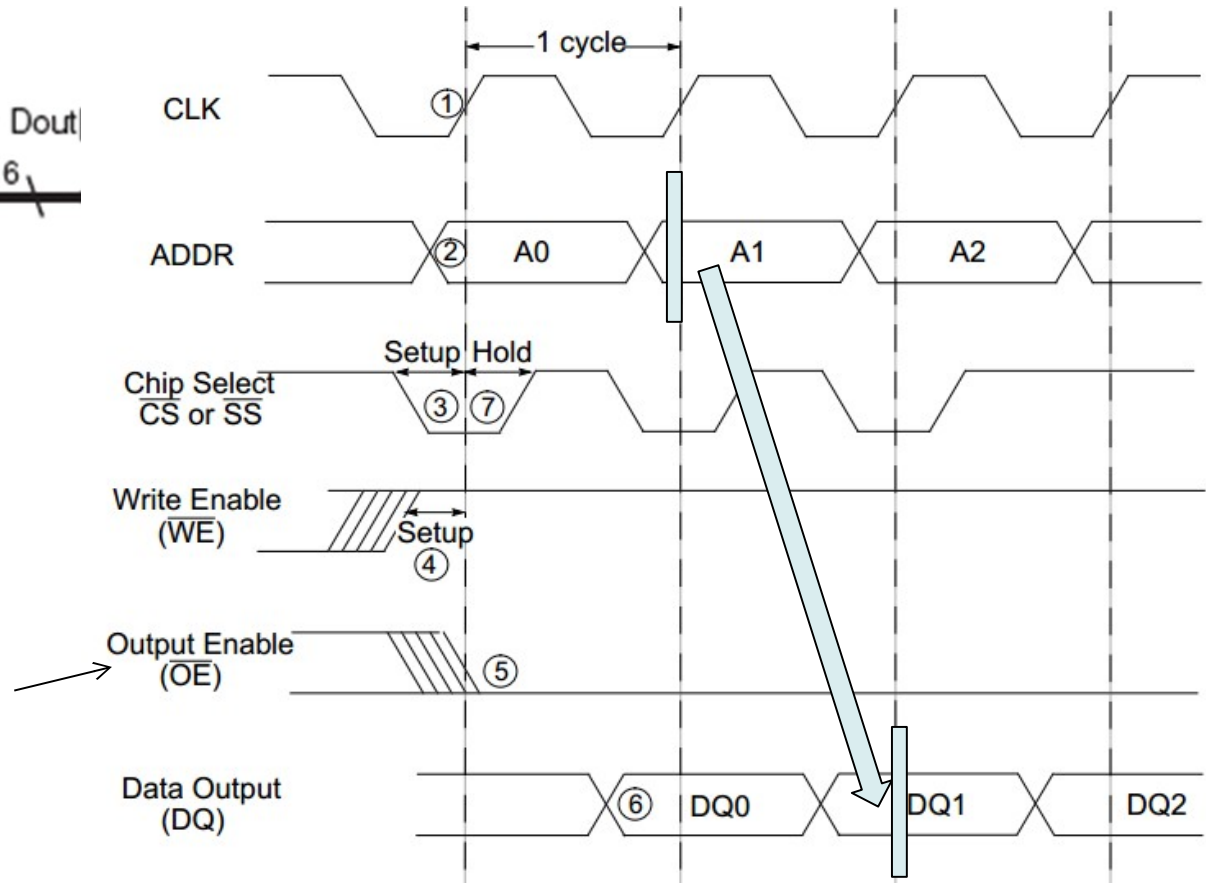


Usual SRAM Chip

Typical synchronous SRAM chip

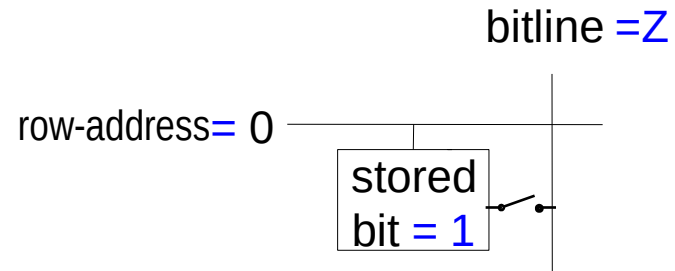
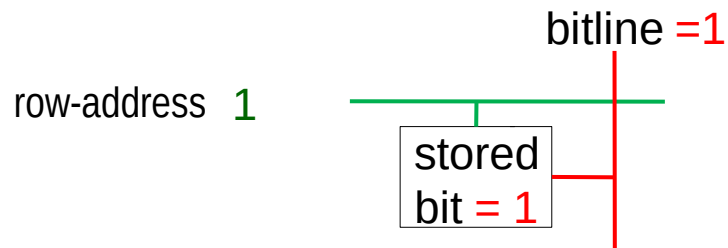
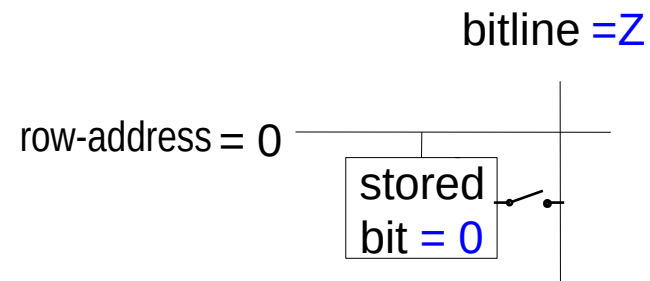
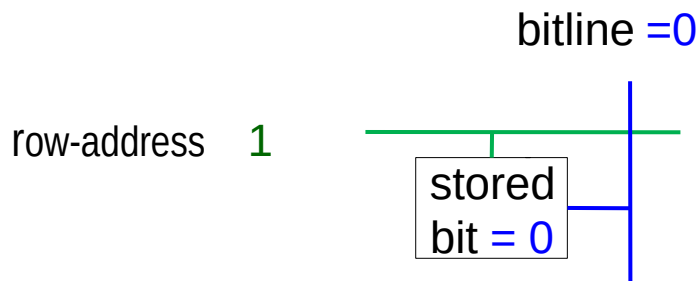
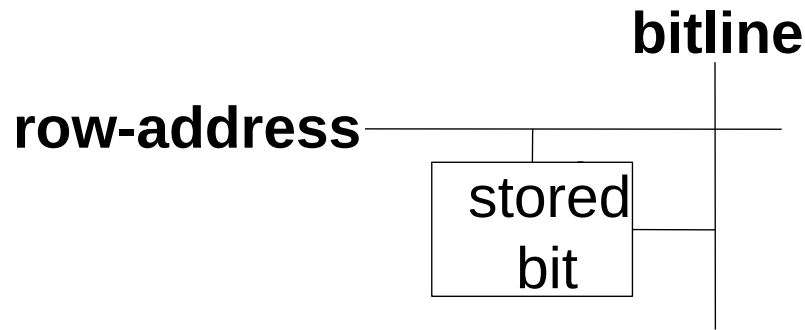


Read example for synchronous case:



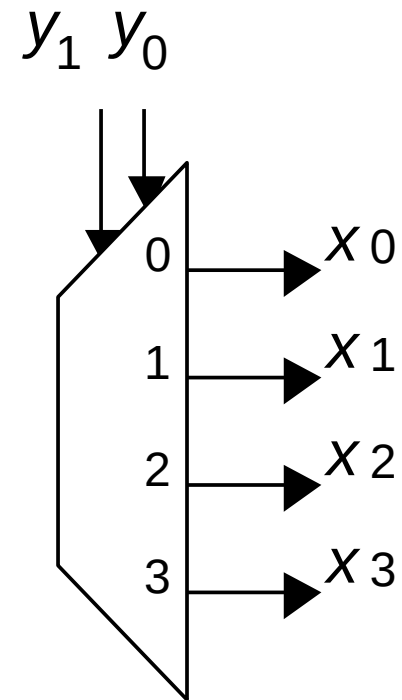
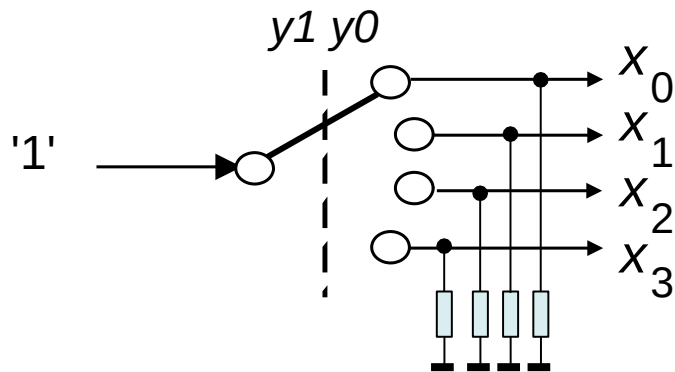
OE signal can be asynchronous

Memory Cell Connection to Matrix



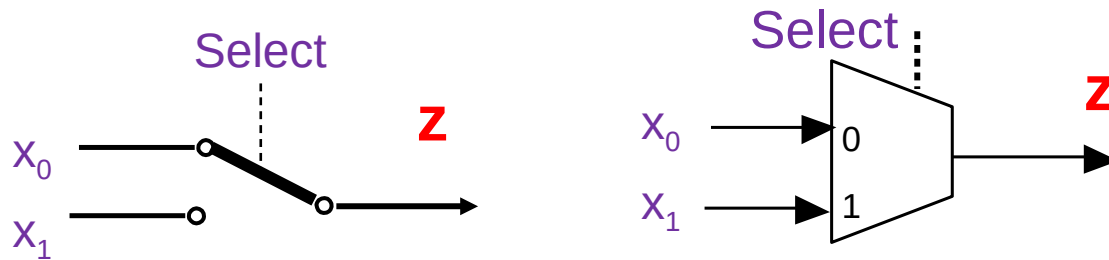
Selector Switch – One from N Decoder

One Hot Decoder *cz: Dekodér 1 ze 4*

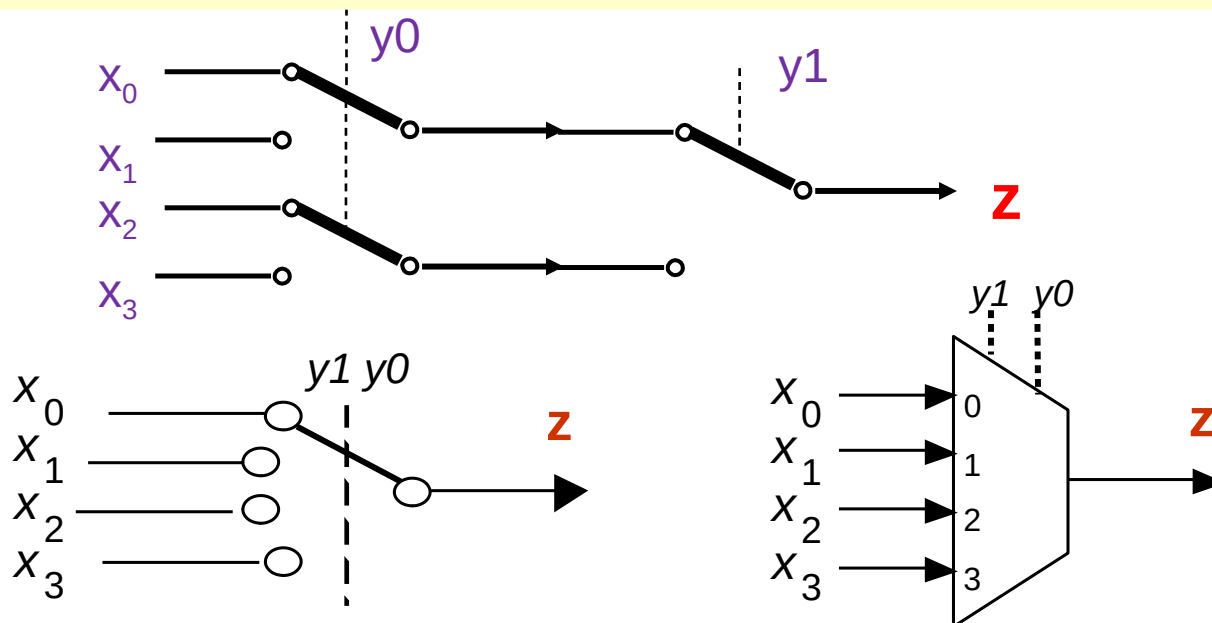


Switch Analogy of Multiplexer

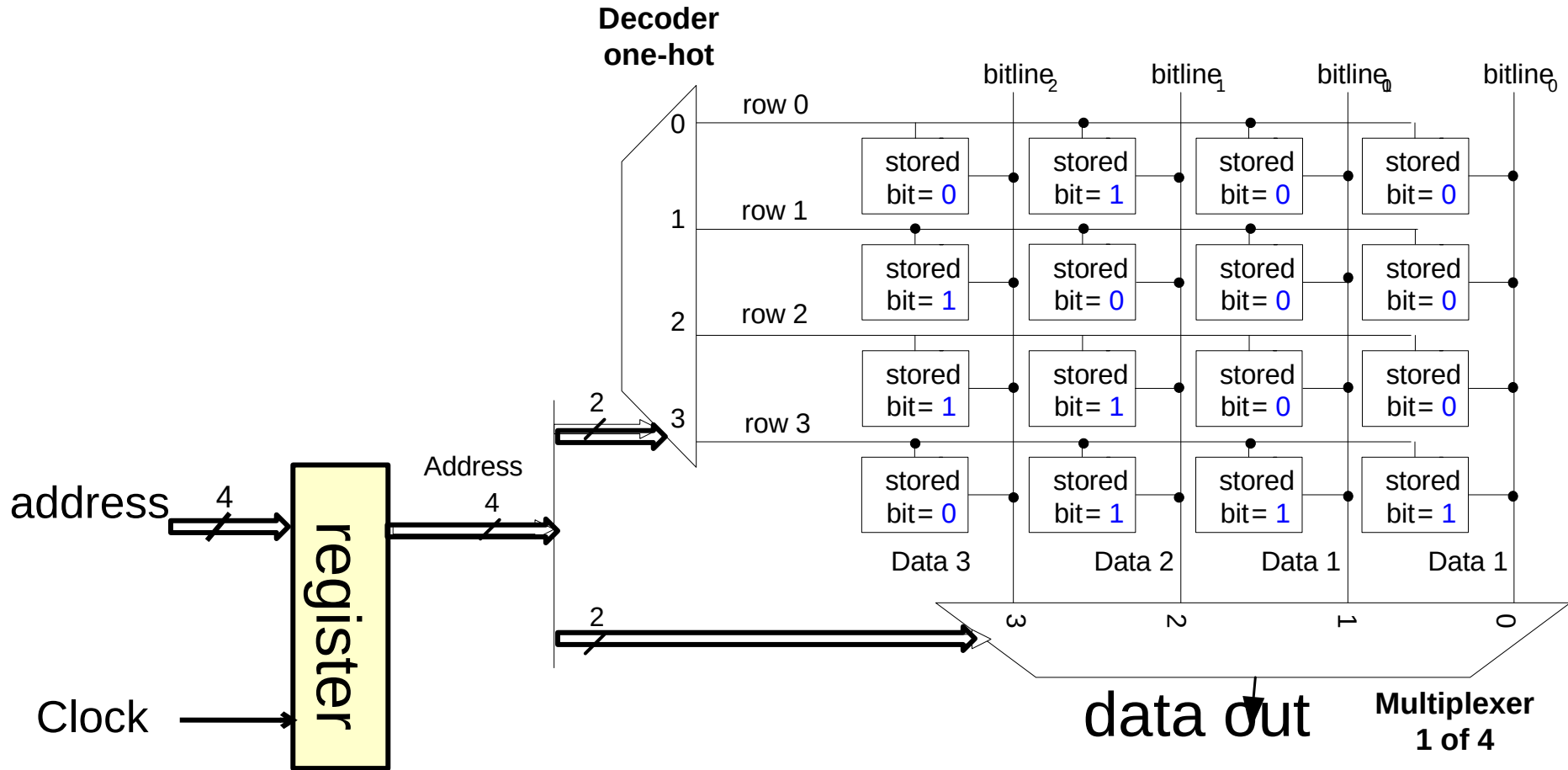
Multiplexer 2 to 1 or 1 of 2 *cz : 2 kanálový (2-vstupový) multiplexor*



Multiplexer 4 to 1 or 1 of 4 *cz : 4 kanálový (4-vstupový) multiplexor*

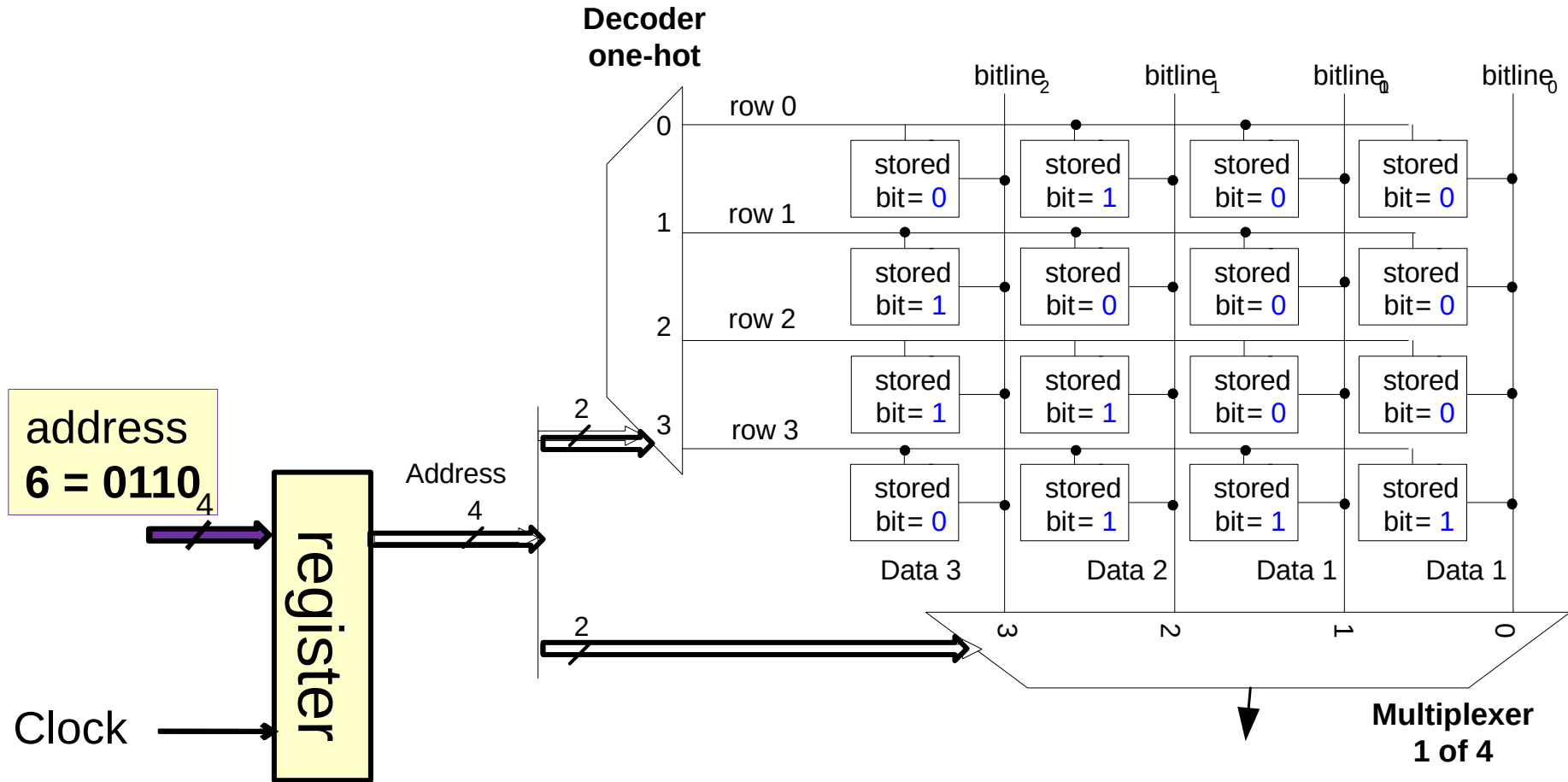


Memory Matrix



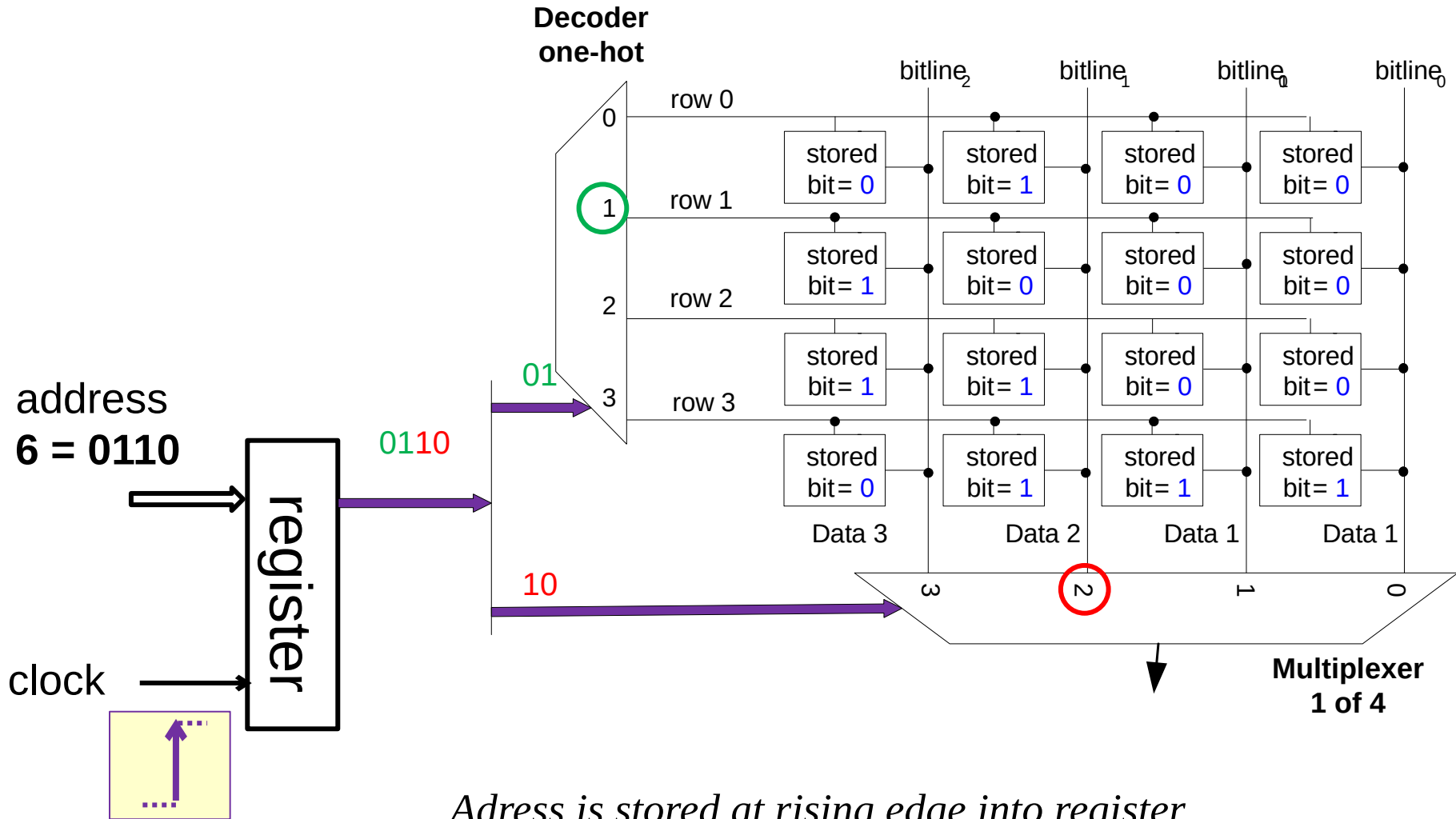
Register is necessary for synchronous memory implementation (SDRAM)

Memory Matrix – Operation



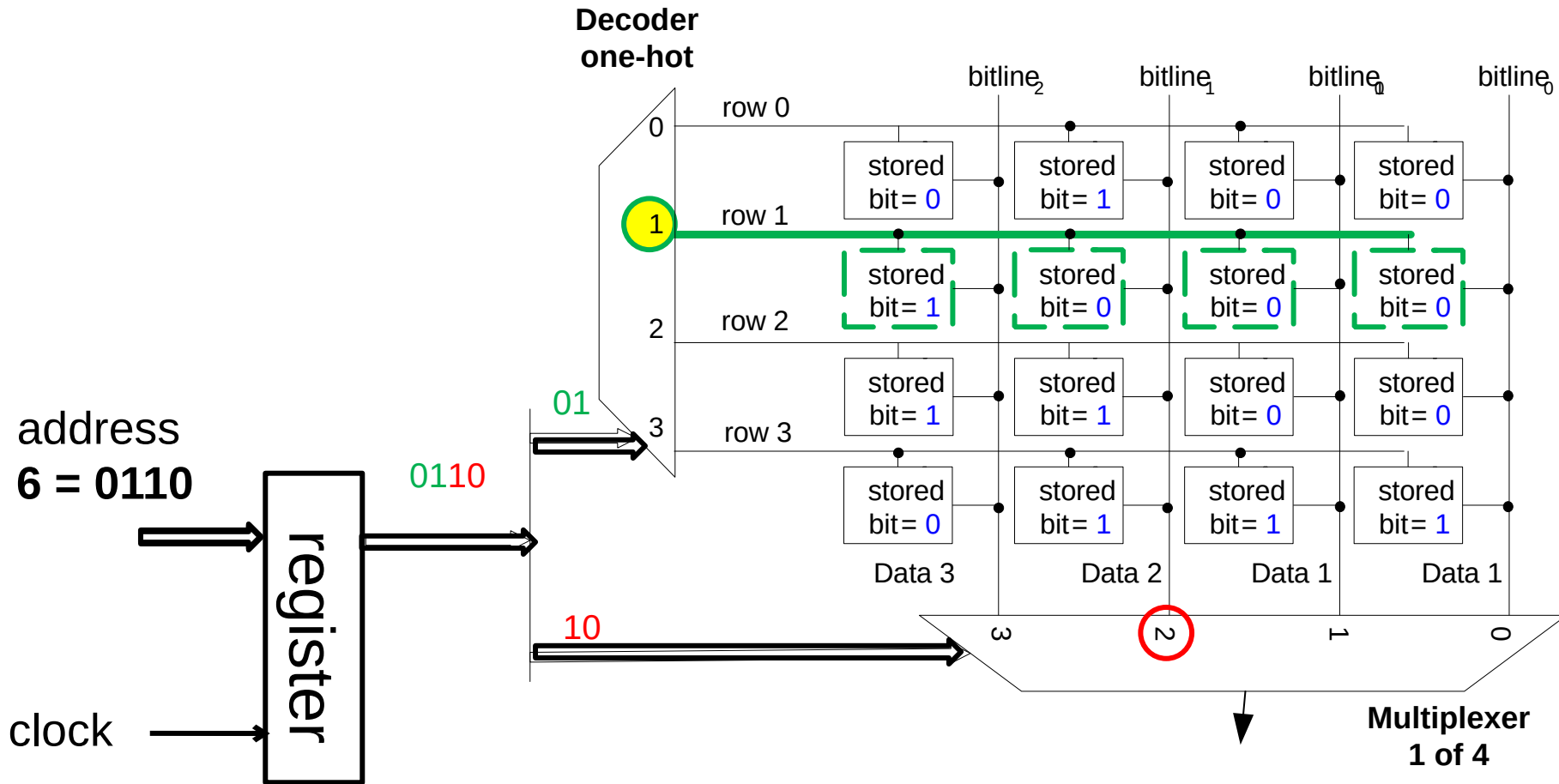
Address is setup at input and it is confirmed by rising edge.

Memory Matrix – Operation



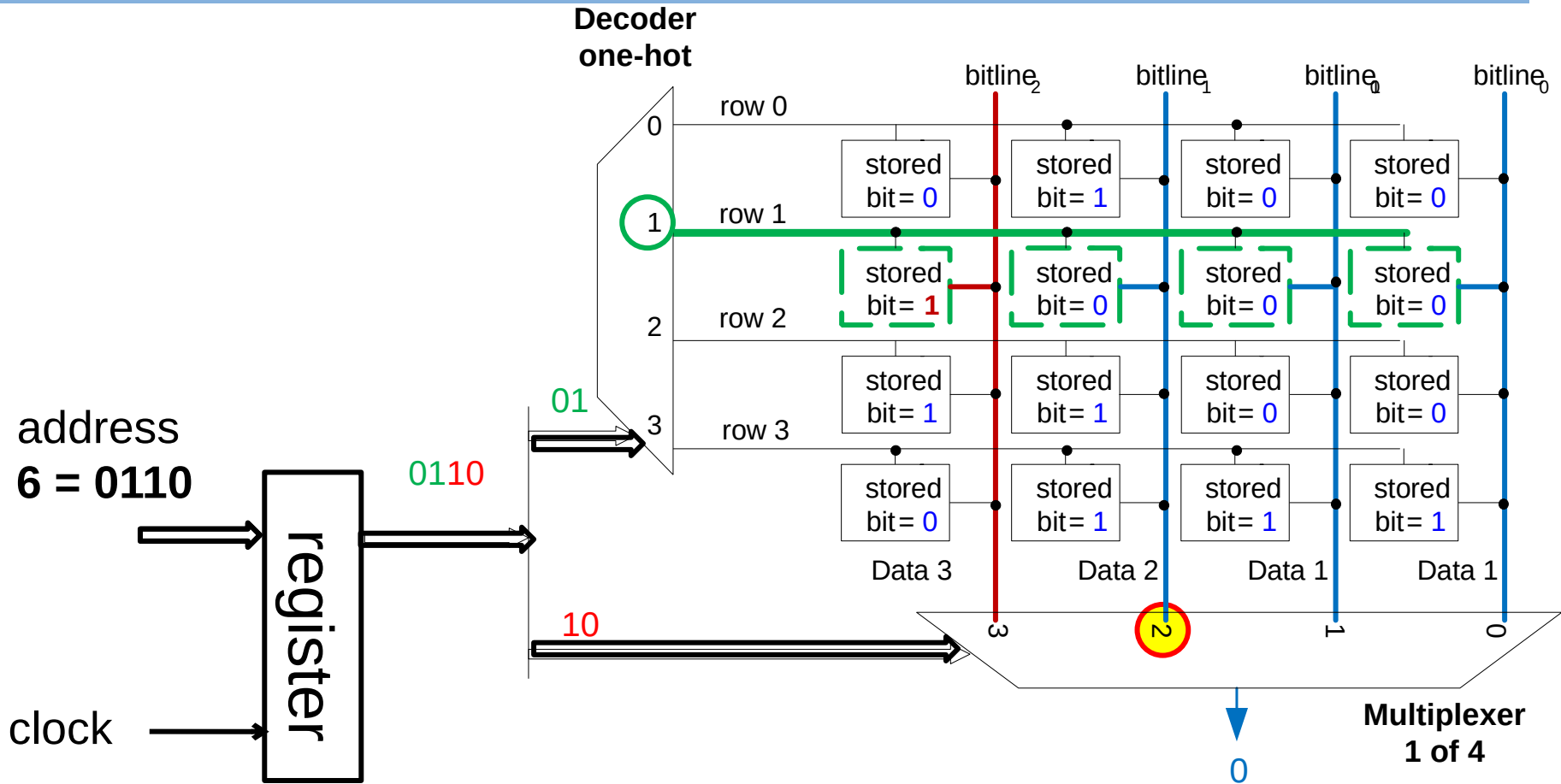
Address is stored at rising edge into register and MSB bits select row and LSB bits column

Memory Matrix – Operation



Decoder activates 1 of N rows and the selected cells are connected to all columns bitlines

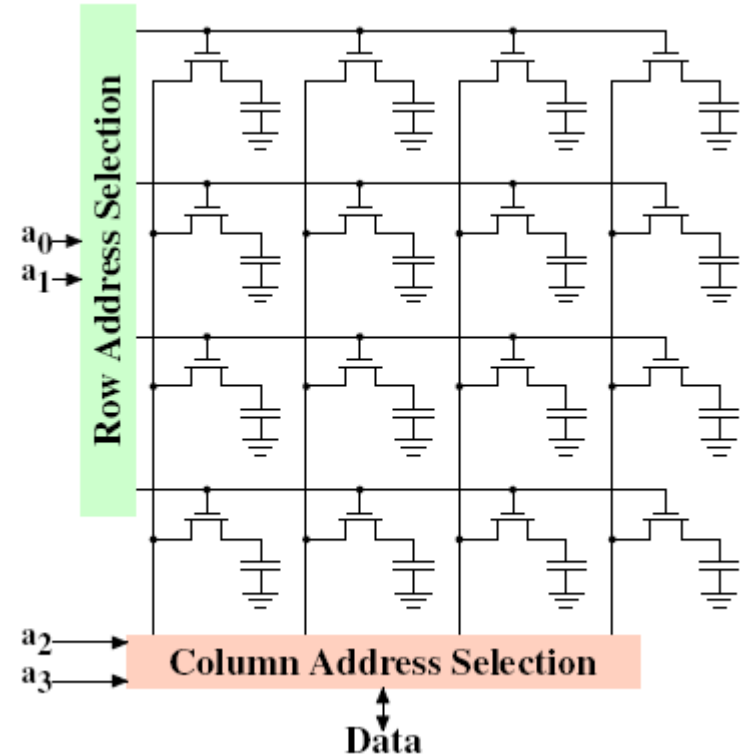
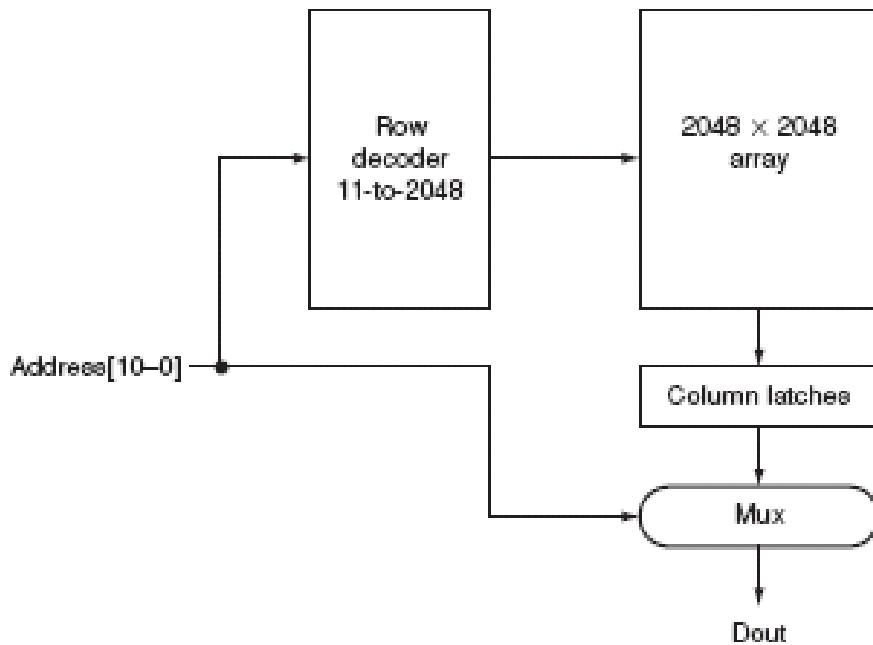
Memory Matrix – Operation



*Multiplexer selects column - **Data 2 = 0***

When register is connected before multiplexer then whole row can be read at once and consecutive data words can be streamed out by multiplexer only switching columns

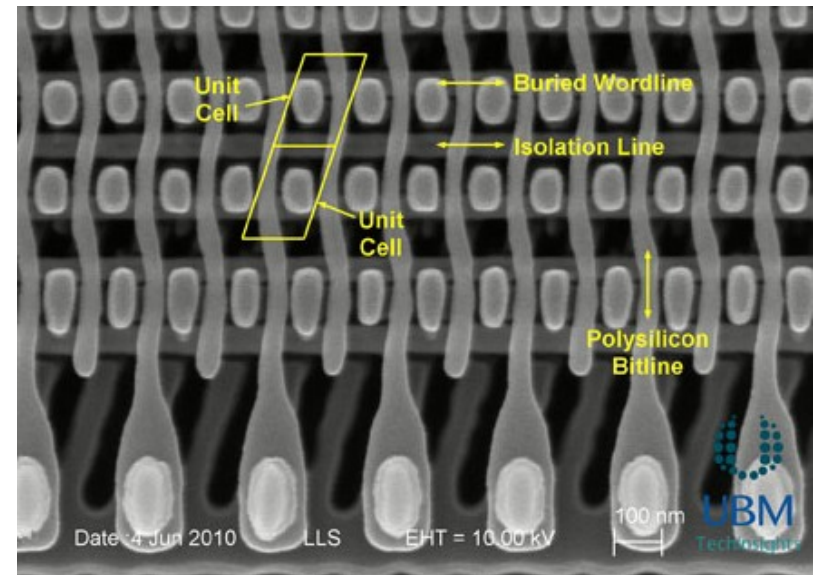
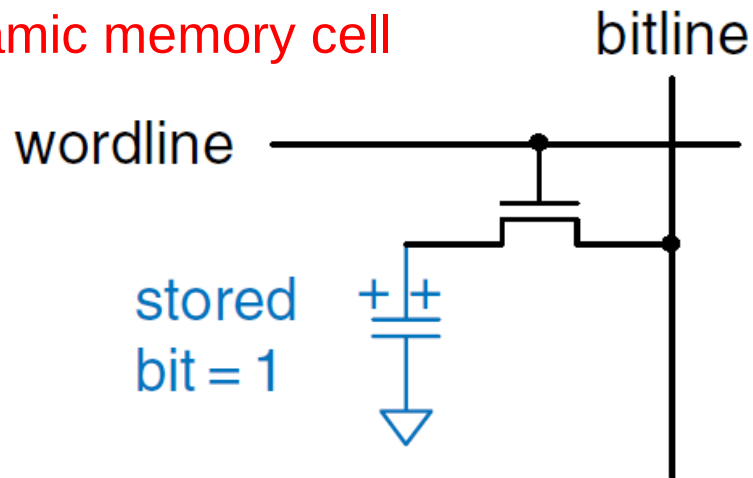
Internal Architecture of the DRAM Memory Chip



This $4\text{M} \times 1$ DRAM is internally realized as an 2048×2048 array of `1b` memory cells

Detail of Dynamic Memory Cell

Single transistor dynamic memory cell



- nMOS transistor nMOS works as analog switch which connects selected cell to „bitline“.
- „wordline“ controls which capacitor is connected to “bitline”

Dynamic Memory Capacitor Parameters

Today DRAM parameters

	Capacity fF [femtofarad]
Capacitor capacity	from 10 fF to 50 fF
Bit line capacity	about 2 fF

[Source: l'INSA de Toulouse]

fF - femtofarad

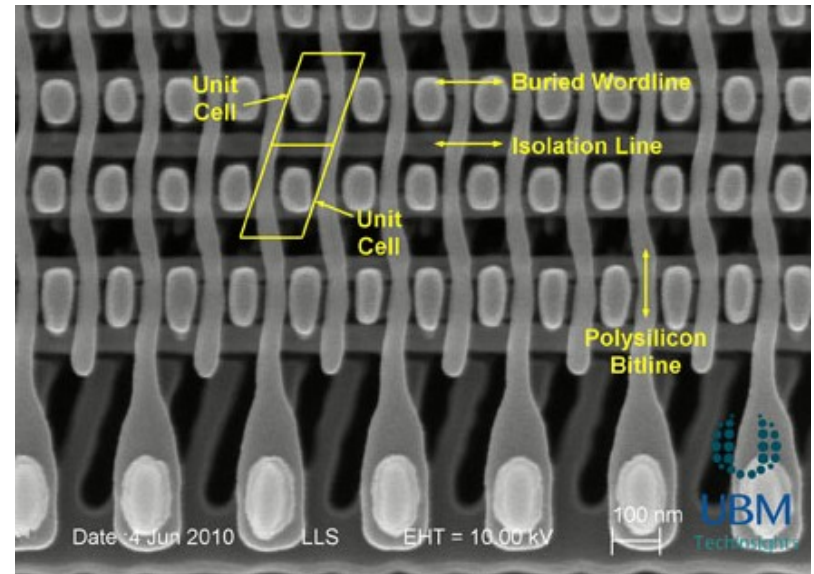
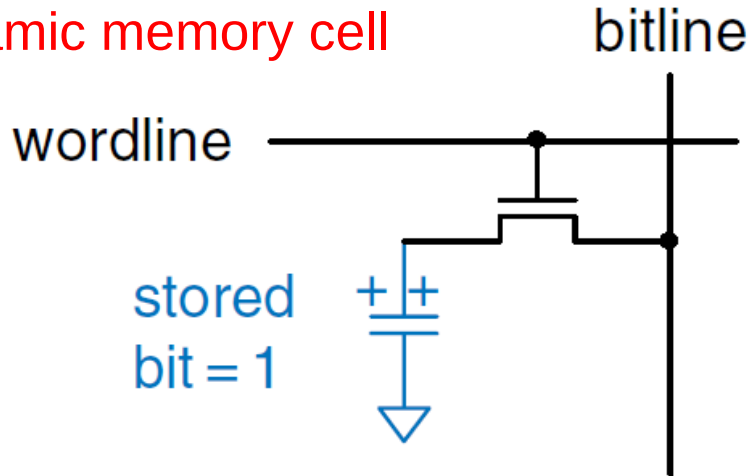
fF is SI unit equal to 10^{-15} Farads.

$$10^{-6} \text{ F} = \mathbf{1 \mu\text{F}} = 10^3 \text{ nF} = 10^6 \text{ pF} = 10^9 \text{ fF}$$

~9 fF is capacity between two plates of 1 mm^2 area with distance between plates around 1 mm,

Detail of Dynamic Memory Cell

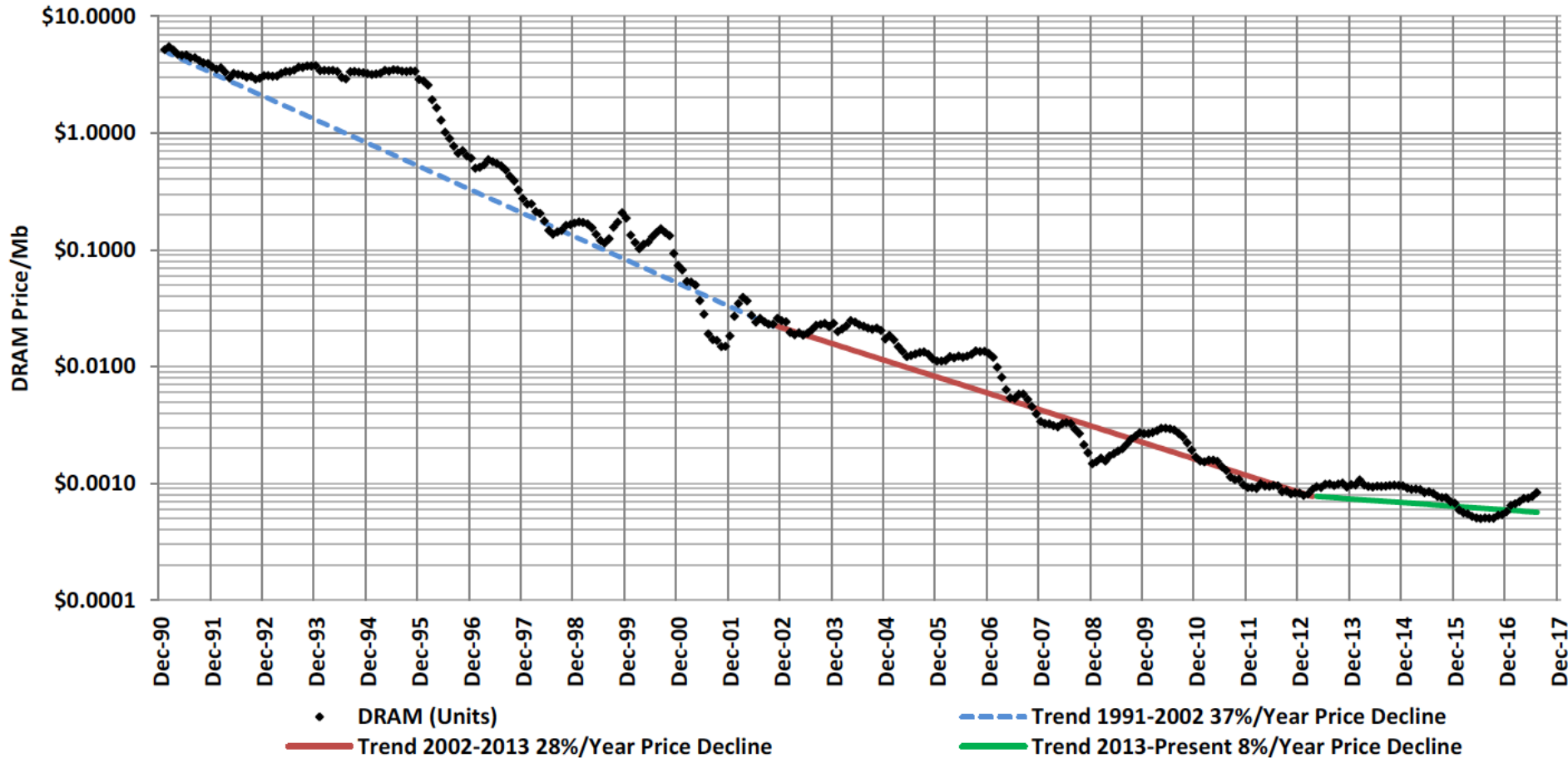
Single transistor dynamic memory cell



- Read operation is complex and **slow**, takes from 20 to 35 ns, and speedup is almost impossible
- Read is **destructive**, capacitor is discharged and original value has to be restored (refreshed) after each read.
- Femto-farad capacitor spontaneously discharges in short time - it is necessary to **refresh** it, in optimum case 60 ms for each cell, but maintenance frequency is multiplied by row count. Required refresh rate depends on temperature

DRAM Memories – Price Seems to Be Settled for Now

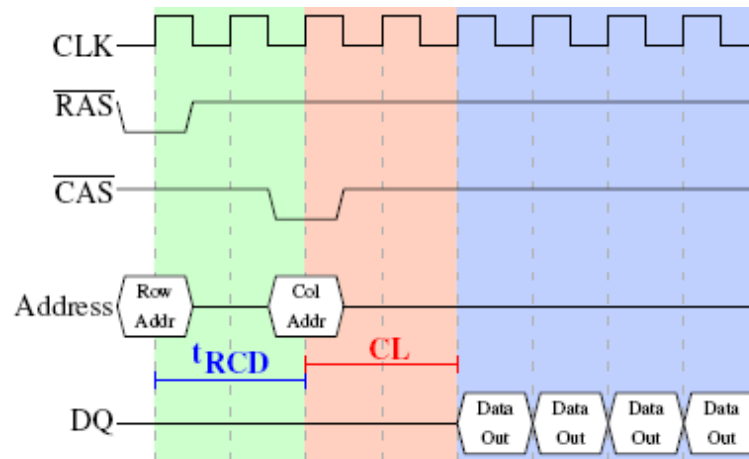
Price for 1 megabit



Source: Wells Fargo Securities, LLC and Semiconductor Industry Association

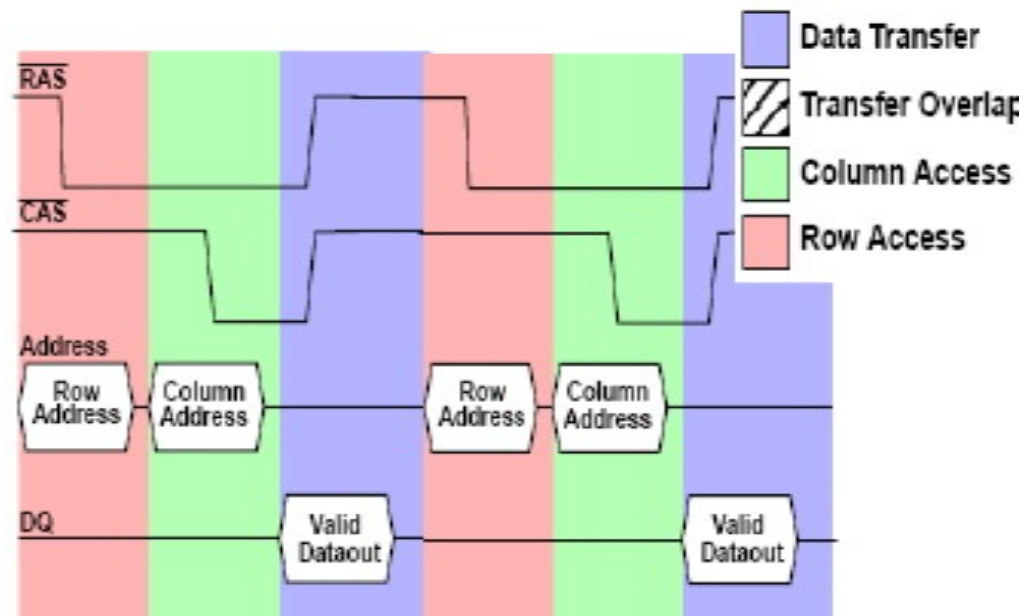
History of DRAM chips development

Year	Capacity	Price[\$]/GB	Access time [ns]
1980	64 Kb	1 500 000	250
1983	256 Kb	500 000	185
1985	1 Mb	200 000	135
1989	4 Mb	50 000	110
1992	16 Mb	15 000	90
1996	64 Mb	10 000	60
1998	128 Mb	4 000	60
2000	256 Mb	1 000	55
2004	512 Mb	250	50
2007	1 Gb	50	40



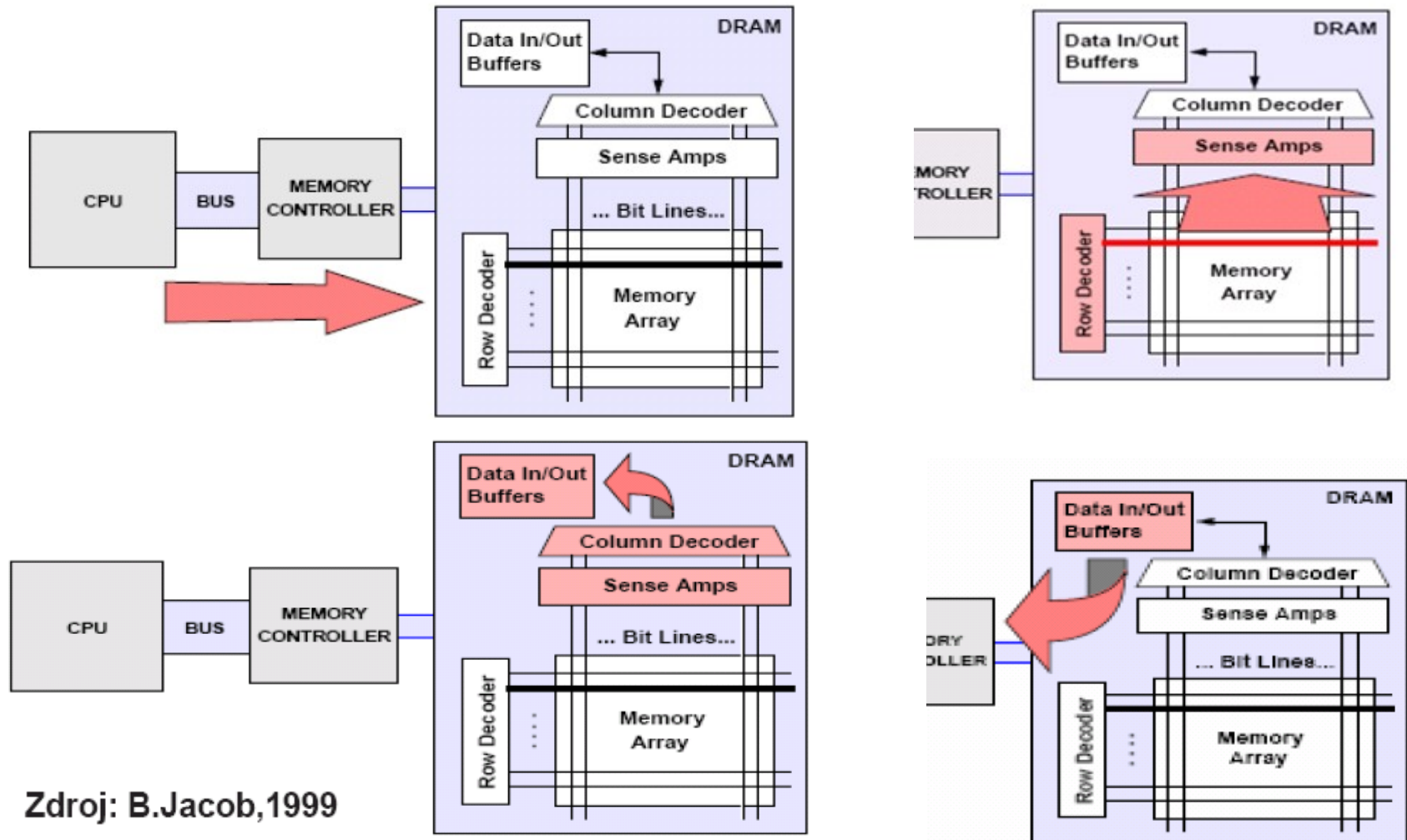
Old School DRAM – Asynchronous Access

- The address is transferred in two phases – reduces number of chip module pins and is natural for internal DRAM organization
- This method is preserved even for today chips



RAS – Row Address Strobe,
CAS – Column Address Strobe

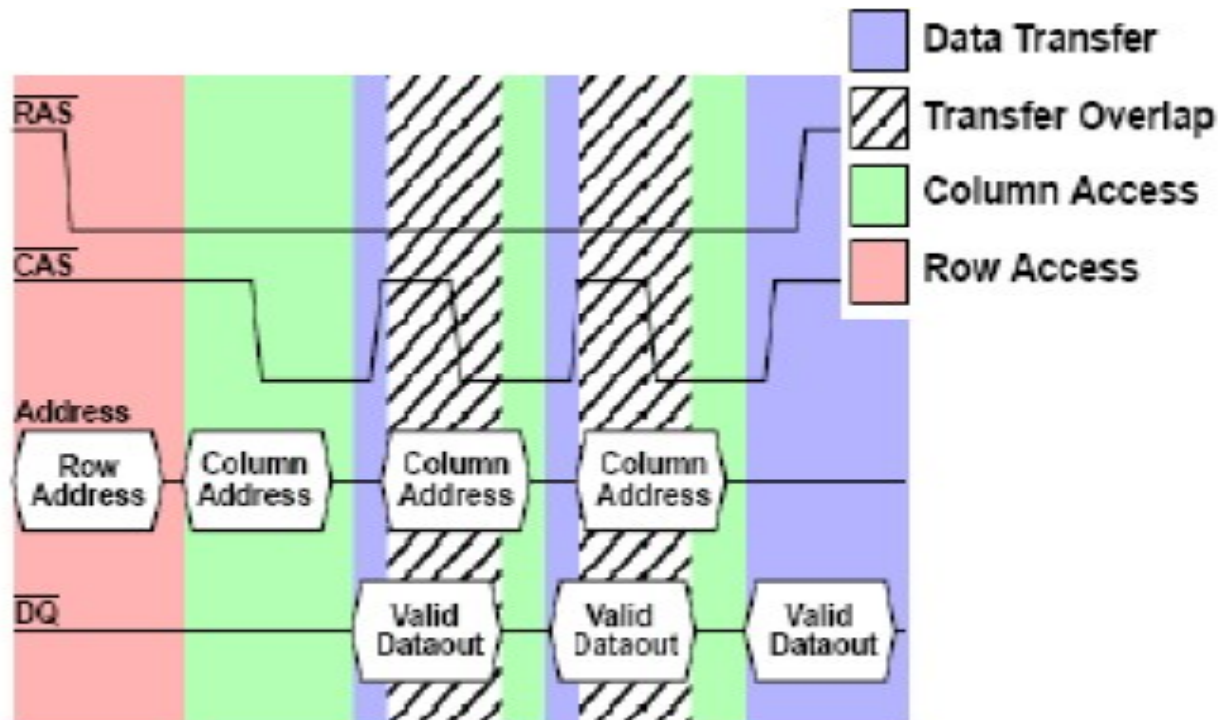
Phases of DRAM Memory Read



Zdroj: B.Jacob,1999

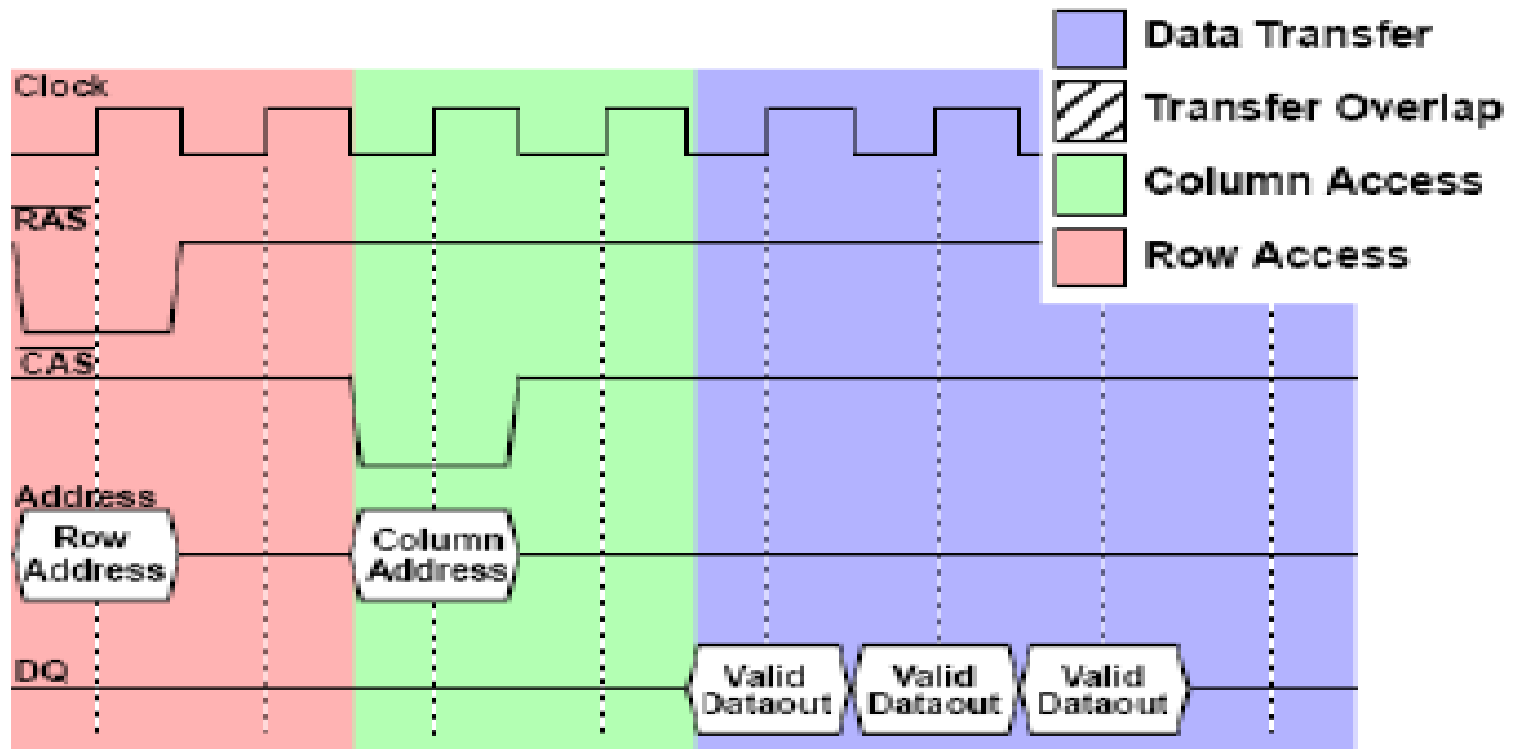
EDO-RAM – About 1995

- Output register holds data during overlap of next read CAS phase with previous access data transfer
this overlap (“pipelining”) increases throughput



SDRAM – end of 90-ties – synchronous DRAM

- SDRAM chip is equipped by counter that can be used to define continuous block length (burst) which is read together



SDRAM – the Most Widely Used Main Memory Technology

- **SDRAM** – clock frequency up to 100 MHz, 2.5V.
- **DDR SDRAM** – data transfer at both CLK edges, 2.5V, I/O bus clock 100-200 MHz, 0.2-0.4 GT/s (gigatransfers per second)
- **DDR2 SDRAM** – lower power consumption 1.8V, frequency up to 400 MHz, 0.8 GT/s
- **DDR3 SDRAM** – even lower power consumption at 1.5V, frequency up to 800 MHz, 1.6 GT/s
- **DDR4 SDRAM** – 1.05 – 1.2V, I/O bus clock 1.2 GHz, 2.4 GT/s
- **DDR5 SDRAM** – expected 2019-2020, ~6 GT/s
- **All these innovations are focused mainly on throughput, not on the random access latency which for large capacities is still 20 to 35 ns.**

Other Main Memory Types

- **QDRx** SDRAM (Quad Data Rate) – not twice as fast, allows only simultaneous read and write thanks to separated clocks for RD and WR, DDR are more effective than QDR for single access type only přístupu.
- **GDDR** SDRAM – today up to GDDR6, designed for graphics cards/GPUs
 - based on DDR memories.
 - data rate accelerated by wider output bus
- High Bandwidth Memory (**HBM**) is a high-performance RAM interface for 3D-stacked SDRAM from Samsung, AMD and SK Hynix.
- Another concept RDRAM (**RAMBUS** DRAM), which use completely different interface. Due to patent litigation are not in use in personal computers from 2003 year.

Notes for Today SDRAMs and Slides

- Use of the banked architecture that enables throughput to be increased by hiding latency of the opening and closing rows. These operations can proceed in parallel on different banks (sequential and interleaved banks mapping). The change result in a minimal pin count increase that is critical for price and density.
- Ulrich Drepper, Red Hat, Inc., What Every Programmer Should Know About Memory